Hiérarchie de contraintes : quelques approches de résolution

Mouhssine Bouzoubaa

THÈSE

présentée pour l'obtention du Diplôme de

DOCTEUR EN SCIENCES
DE
L'ÉCOLE NATIONALE DES PONTS ET CHAUSSÉES

Spécialité: Mathématiques, informatique

par

Mouhssine BOUZOUBAA

SUJET DE LA THÈSE :

Hiérarchie de contraintes: quelques approches de résolution

soutenue à Sophia-Antipolis le 21 Octobre 1996

devant le jury composé de:

Président: René LALEMENT

Rapporteurs : Patrice BOIZUMAULT
Michel RUEHER

Examinateurs: Christian BESSIERE
Bertrand NEVEU
Thomas SCHIEX

à la mémoire de mes parents…

" Demandez le savoir depuis votre naissance jusqu'à votre mort "

Pro. Sid. M.

Remerciements

Mes chaleureux et sincères remerciements vont...

A Bertrand Neveu, directeur de ma thèse. Celui-ci m'a accueilli pendant trois années avec bienveillance dans son équipe, m'a accordé sa confiance et a su me corriger et faire le tri dans tout ce que j'ai pu dire et écrire.

A Taoufik Bouzoubaa, Eric Glemet, Nicolas Chleq, Geir Hasle, Herman Ruge Jervell: des puits de sciences. Ces personnes m'ont toujours poussé à aller plus loin, ils ont su m'écouter et m'encourager à pousser les idées de ce mémoire.

A Patrice Boizumault et Michel Rueher, mes deux rapporteurs, pour leurs conseils avisés et l'intérêt qu'ils ont porté à mon travail en acceptant d'être rapporteurs de cette thèse.

A René Lalement, Thomas Schiex et Christian Bessière qui ont accepté sans aucune réserve d'être membre du jury.

A l'Ilah, Firouz & Hamid Bounafaa, Mhamed Bouzoubaa, Taoufik Bouzoubaa, Rachid Bouzoubaa, Bouchra Bouzoubaa, Amina Bouzoubaa, Hicham Bouzoubaa, Karim Bouzoubaa et toute ma famille : ils m'ont soutenu et aidé pour réaliser cette thèse.

A Dagrun Vikhamar, qui m'a supporté pendant la durée de cette thèse et a su m'aider moralement pour gagner confiance en moi.

A Bernard Larrouturou, humaniste et modeste.

A Stéphane Demphlous, compagnon de chaque jour, humaniste, humouriste, agréable, généreux, conseiller, relecteur, correcteur orthographique, secrétaire, standardiste et enfin grand moraliste.

A Gilles Trombettoni, thésard comme moi, mon colocataire, qui m'a accueilli chez lui la première nuit de mon arrivée dans la région, et pour l'ambiance qu'il a su faire régner durant ces trois années.

A Franck Lebastard, Olivier Corby. Ces personnes m'ont considéré et estimé.

A Saîd & Paula Bouzoubaa, Abdelhay Bounafaa, Youssef & Héléne Bouzoubaa, Majid Rabi, Mohamed Lebar, Mohamed.& Jawad. Diouri, Jean Marc Benoit, Jean Noel Ully, Driss Hadioui, Youssef & Delphine Maklach, Ali Kettani, Khadija Gamisi, Mohamed & Patricia Bennani, pour leur amitié et leur aide qu'ils m'ont apportées lors des moments difficiles.

A Rose Dieng, Michel Jaczynski, Nicolas Prcovic, Bruno Conductier, Maria Cristina Riff, Philippe Ballesta, Olivier Jautzy, Brigitte Trousse, Stéphane Lapalut, Bjorn Sigurd Johansen, Birgit Rognebakke, Per Kristian Nilsen, Philippe Charman et Alain Bernardeau-Moreau pour leurs gentillesse, leur sourire et leur bon coeur.

Enfin, à tous les membres de l'INRIA, du CERMICS et de l'ENPC qui m'ont rendu la vie agréable et qui m'ont accordé leur sympathie (notamment Armel, Claude, Hortense, Sophie, Nathalie, Laetitia, Fabrice, le personnel du restaurant).

Préambule

Ce mémoire de thèse rend compte de travaux effectués au sein de l'équipe Contraintes du CERMICS-INRIA (Centre d'enseignement et de recherche de l'Ecole Nationale des Ponts et Chaussées et l'Institut National de Recherche en Informatique et Automatique) Sophia-Antipolis. L'objet de ces travaux entre dans l'axe de recherche sur les problèmes de satisfaction de contraintes. Le contexte de ces travaux découlait de la question suivante : *et si on est en face d'un problème sur-contraint (pas de solution), que faut il faire ?*

Cette question a lentement mûri après que nous nous soyons *plongé* de nombreux jours dans la lecture et la compréhension de certains articles souvent très intéressants dont quelques uns étaient ardus. Un bref aperçu sur certains de ces articles fait l'objet du **chapitre 1** de notre mémoire.

Pour répondre à cette question, nous nous sommes placé dans le cadre théorique des hiérarchies de contraintes où les contraintes sont fonctionnelles et réparties en contraintes requises et en contraintes de préférences. La satisfaction de ces dernières est gérée par un critère de comparaison entres les configurations (valeurs des variables) du système. Ceci fait l'objet des **chapitres 2** et **3** de notre mémoire.

Plus techniquement, le **chapitre 4** détaille une série de résolveurs qui permettent d'implémenter le modèle théorique de hiérarchies de contraintes et d'obtenir donc des solutions.

Dans le but d'avoir des solutions de meilleure qualité nous avons été amené à concevoir un nouveau résolveur. Ce dernier fait l'objet du **chapitre 5**. Le **chapitre 6** marque des nouvelles définitions d'autres critères de comparaison ainsi que leur intégration dans ce nouveau résolveur.

Deux phases constituent le **chapitre 7** qui marque la deuxième partie de ce mémoire. D'une part, il s'agit de présenter l'aspect théorique de l'intégration des hiérarchies de contraintes dans les Langages de Programmation Logique et d'autre part nous proposons une procédure qui utilise le résolveur décrit dans les **chapitres 5** et **6** pour cette intégration. Nous étudions également son intérêt par rapport à une autre approche.

La troisième et dernière partie est présentée dans **le chapitre 8**. Ici on propose une modélisation d'un résolveur pour la résolution d'une hiérarchie contenant des critères de comparaison différents. Ce résolveur consiste en l'établissement d'un plan de coopération entre résolveurs spécifiques.

1 Introduction

Plusieurs problèmes dans les sciences informatiques doivent gérer les relations entres les objets d'un système. Pour n'en nommer que quelques unes, ces relations peuvent être entre objets graphiques d'une image, peuvent être des relations arithmétiques entre des nombres, une information sur les noeuds adjacents à un noeud dans un graphe, ou ce qui doit être requis dans un travail de planification. Le traitement des relations peut très vite devenir complexe lorsque le nombre d'objets dans le système croît. Ce problème peut être résolu plus facilement lorsque les relations sont représentées d'une façon déclarative c'est-à-dire lorsqu'il y a un langage qui peut capturer les contraintes sous-jacentes sur les objets et ceci indépendamment de la résolution effective des relations. La notion de langage utilisant les contraintes est née de ce désir d'abstraction. Les contraintes forment un outil élégant pour maintenir ce rapport et pour caractériser déclarativement les propriétés entre objets qui peuvent être maintenues par un système sous-jacent.

1.1 Définition d'une contrainte

Intuitivement, on peut considérer que formuler un problème en termes de contraintes revient à définir ce que doit être une solution, c'est-à-dire les propriétés que cette solution doit vérifier, et ce qu'elle ne doit pas être, c'est-à-dire les propriétés qu'elle ne doit pas vérifier.

Une contrainte est donc assimilée à une propriété liant différents objets. Par exemple, la contrainte $x-y=z$ met en relation différents objets (les variables x, y et z) de manière à imposer une restriction sur les valeurs que peuvent simultanément prendre ces objets. Cette contrainte peut être incluse dans un système de contraintes. Dans ce cas, poser la contrainte revient à énoncer une partie du problème.

Une contrainte décrit une relation qui doit être satisfaite. Par exemple, la contrainte de *résistance* dans un circuit électrique consiste à obéir à la loi d'*Ohm*. Les contraintes sont utilisées dans les langages de programmation, les interfaces utilisateur, les logiciels de simulation, et beaucoup d'autres systèmes. Au lieu d'obliger l'utilisateur à écrire et à appeler des procédures pour maintenir la satisfaction d'une relation entre objets, les contraintes permettent de déclarer les besoins de ce maintien [BFW92, BFW94].

En général, les contraintes fonctionnelles sont multi-directionnelles. Par exemple, la contrainte x–y=z peut être utilisée pour calculer la valeur d'une des variables x, y ou z à partir des 2 autres. En général, dans une application donnée, les contraintes sont connectées et le système doit gérer ces connexions de telle sorte que ces contraintes soient toutes satisfaites.

1.2 Modèles de perturbation et de réduction.

Les systèmes décrits dans les langages intégrant la possibilité d'exprimer des contraintes, utilisent une des deux approches suivantes : le modèle de réduction ou le modèle de perturbation. Dans les deux approches, les contraintes sont utilisées pour déterminer les valeurs de leurs variables.

Dans le modèle de réduction, les variables sont initialement non contraintes, les contraintes sont ajoutées, et progressivement le système réduit les valeurs permises des variables. Cette approche est adoptée plus ou moins universellement dans les langages de programmation logique, comme par exemple dans le schéma de langage de Programmation Logique avec Contraintes [Coh90,JL87] et dans les langages de Contraintes Concurrentes [SRP91,Sar89].

Dans le modèle de perturbation, on considère un ensemble de contraintes et des variables ayant des valeurs qui satisfont cet ensemble de contraintes. Si la valeur d'une variable est perturbée, (généralement par une influence extérieure, par exemple une requête d'un utilisateur) la tâche que doit effectuer le système est d'ajuster les valeurs des autres variables afin de resatisfaire l'ensemble des contraintes. Le modèle de perturbation est souvent utilisé dans des systèmes interactifs graphiques basés sur des contraintes comme Sketchpad [Sut63a], ThingLab I [Bor81], Magritte [Gas83], et Juno [Nel85]. Le modèle de perturbation est aussi utilisé dans les systèmes de construction d'interface comme le système Garnet [MGD+90a, MGD+90b].

Généralement, dans le modèle de perturbation, il y a plusieurs façons possibles de rétablir la satisfaction d'une contrainte après une perturbation. Prenons l'exemple trivial de la contrainte x+y=z, et changeons la valeur de la variable y. Plusieurs choix sont possibles pour resatisfaire cette contrainte: on peut changer seulement la valeur de la variable x, ou on peut changer seulement celle de z, ou bien celle de x et de z. Pour limiter ces choix, on peut déclarer des variables en lecture seulement. Dans ces conditions, généralement on utilise les contraintes uni-directionnelles (c.a.d. la contrainte possède une ou plusieurs variables qui sont déclarées comme des variables de lecture). Par exemple, dans la contrainte x+y=z, si les variables x et y sont des variables déclarées comme des variables de lecture, il est clair que lorsque l'utilisateur modifie la valeur de x ou de y le système change celle de z uniquement .

A l'exception des systèmes qui utilisent les contraintes acycliques uni-directionnelles, le problème posé par le modèle de perturbation est qu'il est souvent difficile de déterminer les variables à changer pour resatisfaire les contraintes après une perturbation. Plusieurs techniques ont été proposées dans les premiers systèmes (Sketchpad [Sut63b, Sut63a], Juno [Nel85], IDEAL [Wyk80, Wyk82], Magritte [Gas83], COOL [KK91], Converge[1] [Dis90, Boh90]). Aucune des techniques proposées ne donne entièrement satisfaction, puisque d'une part les solutions trouvées sont non intuitives (non satisfaisantes), et d'autre part ces heuristiques sont intégrées dans le code du système et il est donc difficile de spécifier les solutions préférées et d'altérer ces préférences.

1 Deux versions de ce système ont été développées : une pour la modélisation géométrique en 3D [Sis90] et la seconde pour l'extraction de graphe cyclique.

1.3 CSP et flexibilité

La résolution d'un problème de satisfaction de contraintes (CSP) équivaut à partitionner l'ensemble des valuations en deux ensembles : un qui contient les solutions du problème et un autre qui contient son complémentaire [SG92]. Dans les problèmes sur-contraints, à savoir ceux qui n'ont aucune solution alors que l'utilisateur en veut une, la prise en compte des préférences permet de relâcher certaines contraintes et de donner ainsi des solutions (celles qui seront jugées les moins mauvaises). Le besoin d'étendre le formalisme des CSP pour pouvoir considérer les préférences entres contraintes, s'est fait ressentir depuis quelques années [Far94]. Plusieurs tentatives d'extensions ont été proposées. On trouve à la fois des tentatives de définition de nouveaux modèles et formalismes et des travaux plus techniques proposant des algorithmes.

Les préférences peuvent être exprimées plus ou moins localement c'est-à-dire :

- Sur des valeurs ou des combinaisons de valeurs possibles pour une variable ou un n-uplet de variables.

- Sur des contraintes : il s'agit d'exprimer l'importance de satisfaire chaque contrainte.

- Directement entre solutions : il s'agit d'une mesure évaluant la qualité d'une solution.

Le premier cas est l'idée maîtresse à l'origine de plusieurs travaux. Ce cas est apparu très tôt dans la littérature liée aux CSP: en 1975, Waltz dans [Wal75] fait une distinction entre instanciation "souhaitable" d'une contrainte et instanciations "moins souhaitables". Ces dernières ne sont considérées que si nécessaire. En 1983, cette idée a été reprise dans le domaine de la recherche opérationnelle par Hummel et Zucker dans [HZ83]. Plus récemment, des travaux intègrent cette idée et optent pour une représentation des contraintes flexibles sous forme d'ensemble flou, nous citons ici les travaux de Freuder et Snow, Bowen et Lai et Bahler, Guan et Friedrich, Martin-Clouaire, Faltings et Haroud et Smith respectivement dans [FS90], [BLB92], [GF92], [Clo92] et [FHS92].

Dans le deuxième cas, la flexibilité réside dans la capacité à ne pas satisfaire des contraintes de moindre importance quand elles sont conflictuelles. Plusieurs travaux sur des CSP à domaine fini considèrent ce cas. Par exemple, Jussien et Boizumaut dans [JB96] considèrent au départ que toutes les contraintes sont aussi importantes les unes que les autres et proposent un système de relaxation de contraintes sur les CSP sur-contraints (il n'y a pas de solution qui satisfait l'ensemble des contraintes) qui détermine la (ou les) contrainte(s) à supprimer pour aboutir à une solution. Ce système de relaxation est basé en partie sur le maintien de déductions qui s'inspire de l'algorithme de recherche de solution utilisant les ensembles de contraintes non-consistants trouvés au cours de la recherche. Cet algorithme est proposé par Schiex et Verfaillie dans [SV94, VS95].

Pour Schiex dans [Sch92] une priorité est associée à chaque contrainte, et les meilleures solutions sont celles qui minimisent la priorité de la plus importante des contraintes violées. Un ordre basé sur l'inclusion est un raffinement de celui de Schiex a été indiqué par Brewka dans [Bre89]. Dans cet ordre, deux valuations seront départagées si l'ensemble des contraintes satisfaites par l'une est inclus dans l'ensemble des contraintes satisfaites par l'autre. Cet ordre est celui obtenu par un critère de comparaison local (*Localement-Prédicat-Meilleur)* défini par Alan Borning dans [BDF+87]. Descottes et Latombe dans [DL85] proposent un raffinement de celui-ci, l'ordre lexicographique. On retrouve également cet ordre chez Alan Borning et Thomas Schiex dans [BDF+87, BFW92, BFW94, SFV95]. Freuder dans [FS90] considère que toutes les contraintes peuvent être violées et que les meilleures solutions sont celles qui minimisent le nombre de contraintes violées. Mohr et Masini dans [MM88] proposent de rechercher une instanciation telle que, pour chaque variable, il y ait au moins une certaine proportion de contraintes satisfaites.

Borning et son équipe dans [BDF+87, BFW92, BFW94] ont identifié des formes de préférence entres contraintes qui prennent en compte le deuxième cas ou le troisième (les distances aux contraintes sont considérées comme des mesures qui évaluent les qualités des solutions). Ces travaux portent sur la combinaison multi-critères et permettent d'identifier les principaux modes de combinaison des contraintes avec priorité (la minimisation de la priorité de la moins satisfaite des contraintes, le raffinement basé sur l'inclusion (*Localement-Prédicat-Meilleur*), l'ordre lexicographique et la maximisation du nombre de contraintes satisfaites (*Globalement-Meilleur*)).

1.4 Définition d'une hiérarchie de contraintes

Une hiérarchie de contraintes peut être vue comme une solution au problème de spécification déclarative de ce qu'il faut changer à la suite d'une perturbation introduite sur une contrainte du système [BDF+87, BFW92, BFW94]. Dans une hiérarchie de contraintes, le programmeur ou l'utilisateur peut exprimer deux types de contraintes. Des contraintes requises (aussi appelées des contraintes dures) et des contraintes de préférence (aussi appelées des contraintes flexibles). Les contraintes requises doivent être satisfaites. Le système doit essayer de satisfaire les contraintes de préférence si possible, mais aucune erreur n'est générée, dans le cas où le système ne peut pas les satisfaire. Les contraintes de préférence sont réparties en un certain nombre de niveaux hiérarchiques. Les niveaux sont ordonnés selon l'importance des contraintes de préférence qu'ils contiennent. Par exemple, on peut exprimer trois contraintes unaires pour la relation $x+y=z$: deux contraintes unaires de poids fort qui consistent à dire que les variables x et y seront non-modifiables, et la troisième contrainte unaire est de poids faible et qui consiste à dire que la variable z est non-modifiable. Etant donnée cette hiérarchie et à la suite d'une perturbation qui consiste à changer la valeur de la variable x, le système va tenter de resatisfaire la relation $x+y=z$ en modifiant la valeur de la variable z au lieu de modifier celle de la variable y.

Les hiérarchies de contraintes ont plusieurs domaines d'applications (tout genre d'application où l'on veut prendre en compte les contraintes de préférence), comme par exemple dans la planification, l'ordonnancement ou encore la manipulation de figures géométriques. Un exemple simple serait de considérer le problème de poser un tableau dans un document [BFW94], on aimerait bien que le tableau soit dans une seule page, tout en laissant le même espace inter-ligne. Ce problème peut être représenté par une interaction entre deux contraintes. La première est une contrainte requise qui serait : les hauteurs entre deux lignes quelconques du tableau sont égales et strictement supérieures à zéro. La deuxième est une contrainte de préférence qui serait : le tableau entier loge dans une page. Considérons un autre exemple : supposons que l'on déplace par la souris une partie d'une figure géométrique contrainte de rester sur l'écran. Lorsque cette partie est en mouvement, d'autres parties de la figure veulent se déplacer aussi pour garder la satisfaction des contraintes de cette figure. Cependant, si les positions de ces autres parties de la figure vont être en dehors de l'écran, on préférerait qu'elles restent à leurs positions initiales au lieu de se déplacer aussi. De plus, il peut y avoir un indicateur sur les parties qui peuvent se déplacer et celles qui doivent rester fixes. Dans ce cas, l'utilisateur peut avoir des préférences et donc la hiérarchie de contraintes est utilisée pour satisfaire ce désir.

1.5 Contribution

Nous nous sommes intéressés au formalisme des hiérarchies de contraintes définies par Alan Borning, où les contraintes sont réparties par niveaux d'importance. Une configuration est une valuation de l'ensemble des variables. On définit un ordre partiel sur les configurations à partir des erreurs, qui peuvent être binaires (satisfaction/non-satisfaction d'une contrainte) ou numériques et de deux mécanismes d'agrégation de ces erreurs :

- dans chaque niveau, les erreurs de la configuration sur toutes les contraintes du niveau sont combinées,

- globalement, ces combinaisons sont agrégées sur tous les niveaux. Les solutions sont les valuations maximales pour cet ordre partiel.

Dans ce cadre, la plupart des travaux sur des algorithmes (*Blue, DeltaBlue, SkyBlue, QuickPlan*) basés sur la propagation locale pour les hiérarchies de contraintes utilisent des erreurs binaires et un mode de comparaison local assez simple entre les différentes configurations :

une valuation (instanciation de l'ensemble des variables) sera meilleure qu'une autre s'il existe un niveau dans lequel elle satisfait un sur-ensemble des contraintes de la seconde, et si pour tous les niveaux plus importants elles satisfont toutes deux les mêmes contraintes.

Pour affiner cet ordre partiel, nous avons conçu un nouveau résolveur de propagation locale [Bou94,Bou95a,Bou95b,BN96] qui prend en compte différents modes de combinaison des erreurs par niveau et utilise une agrégation globale de type lexicographique sur les valeurs de ces combinaisons. Potentiellement, cette agrégation globale permet d'obtenir des solutions meilleures que celles obtenues en utilisant un mode de comparaison local entres les différentes configurations.

Nous avons en particulier intégré dans ce nouvel algorithme les trois modes suivants :

- le nombre de contraintes non satisfaites [Bou95c, BNH95a],

- la somme des poids des contraintes non satisfaites, un poids étant attaché à chaque contrainte [BNH96],

- une combinaison où les poids représentent des priorités : on préférera une configuration satisfaisant les contraintes de poids fort, les contraintes de poids plus faible étant considérées comme des contraintes de remplacement [BNH95b].

Nous nous sommes intéressés aussi à la Programmation Logique par Hiérarchie de Contraintes (*PLHC*). Cette dernière étend la Programmation Logique par Contraintes (*PLC*) en incluant les hiérarchies de contraintes. Des travaux précédents dans ce domaine décrivent des prototypes basés sur un mode de comparaison local [Wil92, WB89, BMM+89, BDF+87]. L'expérience d'écrire des programmes avec ces prototypes montre que ce mode de comparaison local donne des solutions non intuitives. Ceci résulte de la restriction de ce mode local de comparaison à ne pouvoir comparer que des solutions résultant d'une seule hiérarchie (connu par l'appellation comparaison intra-hiérarchie). Cependant, plusieurs applications pratiques en *PLHC* suggèrent le besoin d'une comparaison entre les solutions résultant des différents choix de règles du programme afin d'éliminer celles qui sont non intuitives (c.à.d. jugées moins bonnes) (connu par l'appellation comparaison inter-hiérarchies). Cette comparaison ne peut se réaliser que par l'utilisation d'un mode de combinaison utilisant une agrégation globale. Notre contribution ici a été la conception d'un algorithme [Bou96] basé sur notre premier algorithme et sur le fondement théorique de comparaison inter-hiérarchies.

Enfin, nous nous sommes intéressés aussi au cas où les modes de combinaison peuvent varier selon les niveaux. Par exemple, dans une hiérarchie contenant trois niveaux d'importance, on peut utiliser pour les deux premiers niveaux un critère local pour lequel une solution sera meilleure qu'une autre si elle satisfait un sur-ensemble des contraintes satisfaites par la seconde. Pour le troisième niveau, un comparateur plus fin, comme la somme des carrés des erreurs, qui impose un ordre total sur les configurations, peut être utilisé pour départager les solutions. Ce troisième niveau, utilisant des erreurs numériques, ne peut plus être résolu par propagation locale. La résolution d'une tel type de hiérarchie nécessite plusieurs résolveurs spécifiques aux types des contraintes de cette hiérarchie ainsi qu'un plan de coopération entre ces résolveurs spécifiques. Une approche prometteuse est l'utilisation des techniques de la propagation locale pour partitionner un système de contraintes en des sous-ensembles de contraintes pouvant êtres résolus par ces résolveurs spécifiques. Des travaux précédents sur ce problème dans [HKS+94] avaient abouti à la réalisation d'un résolveur pour des contraintes dont les méthodes ne possèdaient qu'une seule variable de sortie. Notre recherche s'est orientée vers la généralisation de ce résolveur pour pouvoir prendre en compte des contraintes possédant des méthodes recalculant plusieurs variables à la fois [Bou95d].

1.6 Plan de lecture

Le chapitre 2 présente les définitions formelles des contraintes, hiérarchies de contraintes, et les solutions à une hiérarchie de contraintes. Il inclut également des exemples d'utilisation des hiérarchies de contraintes et des propriétés sur des différents comparateurs qui servent à ordonner les solutions d'une hiérarchie de contraintes.

Le chapitre 3 présente les extensions de la théorie des hiérarchies des contraintes. Ces extensions incluent les notions suivantes: *lecture-seulement*, *écriture-seulement*, fonctions objectifs, comparaison inter-hiérarchie, solution type.

Le chapitre 4 présente les fonctionnalités d'une série d'algorithmes de propagation locale existants pour la résolution des hiérarchies de contraintes. Chacun de ces algorithmes est basé sur un comparateur de la théorie des hiérarchies de contraintes présentée au chapitre 2.

Le chapitre 5 marque la première partie de notre travail portant sur la conception d'un nouvel algorithme (nommé Houria) pour la résolution d'une hiérarchie de contraintes fonctionnelles. Dans le but d'obtenir des solutions de meilleure qualité, ce nouvel algorithme est basé sur un critère de comparaison global.

Le chapitre 6 présente la généralisation de l'algorithme présenté au chapitre 5. C'est ainsi que l'on définit deux autres critères de comparaison qui sont intégrés dans l'algorithme conçu. On présente également dans ce chapitre des mesures de performance de cet algorithme effectuées sur des problèmes générés aléatoirement. Ce chapitre compare au point de vue fonctionnel cet algorithme et les algorithmes existants.

Le chapitre 7 marque la deuxième partie de notre travail et présente un aperçu sur l'extension de la théorie des hiérarchies de contraintes concernant son intégration dans le monde de la Programmation Logique. Ce chapitre décrit un algorithme pour la comparaison inter-hiérarchie de solutions résultant des différents choix de règles d'un programme exprimé dans le paradigme *HCLP*. Cet algorithme est basé sur l'algorithme décrit dans les chapitres 5 et 6 et sur l'extension de la théorie des hiérarchies de contraintes.

Le chapitre 8 marque la troisième partie de notre travail portant sur la généralisation d'un résolveur existant qui pilote différents résolveurs. Cette généralisation présente des nouvelles définitions et une extension du mécanisme initial utilisé, ceci ayant pour but de pouvoir prendre en compte des contraintes possédant des méthodes recalculant plusieurs variables de sortie à la fois.

2 Contraintes et hiérarchies de contraintes

Dans ce chapitre, nous présentons un état de l'art sous la forme de définitions formelles des contraintes, des hiérarchies de contraintes, et des solutions à une hiérarchie de contraintes [BFW94,Wil92, HKS+94]. Nous incluons également des exemples d'utilisation des hiérarchies de contraintes tirés de [BFW92, WB89, FWB92] et des propriétés sur différents comparateurs qui servent à ordonner les solutions d'une hiérarchie de contraintes [BFW92, BFW94].

2.1 Définitions

Une contrainte est une relation entre variables sur un domaine D (c.a.d. entiers, réels, booléens, domaine fini). Une contrainte étiquettée, notée ec, est une contrainte c avec une étiquette e qui exprime l'importance de la contrainte. Nous associons des noms symboliques aux différentes étiquettes des contraintes. Dans la suite de ce chapitre, nous associons à chacun de ces noms symboliques un entier compris entre 0 et n, où n est le nombre de niveaux de la hiérarchie. Le premier niveau 0 est associé au nom symbolique *requise*. Ce niveau de la hiérarchie est toujours réservé aux contraintes dures (c.a.d. absolues, requises).

Une hiérarchie de contraintes est un ensemble de contraintes étiquetées. Etant donnée une hiérarchie de contraintes H, H_0 dénote les contraintes requises dans H. Dans le même esprit, les ensembles H_1, ... H_n sont associés respectivement aux niveaux 1, ...n sont définis.

Une solution à la hiérarchie de contraintes H est une valuation pour les variables dans H (c.a.d. une fonction qui associe à chaque variable dans H un élément de D). On désire définir l'ensemble S qui contient toutes les solutions de H. Plus clairement, chaque valuation dans S doit satisfaire toutes les contraintes dures. De plus, elle doit satisfaire aussi bien que possible l'ensemble des contraintes de préférences en respectant leurs étiquettes relatives. Pour formaliser ce souhait, on définit en premier l'ensemble de valuations noté S_0 tel que chaque valuation dans S_0 satisfasse l'ensemble des contraintes dans H_0. Après et en utilisant S_0, on définit l'ensemble S en éliminant les valuations qui sont moins "bonnes" que d'autres dans S_0. Cette élimination est établie par l'utilisation d'un prédicat "*meilleur*" qui est un comparateur .

$S_0 = \{ \theta: \forall c \in H_0, c\,\theta = vrai \}$ et $S = \{ \theta: \theta \in S_0 \wedge \forall \eta \in S_0 \neg meilleur(\eta, \theta, H) \}$

où (c θ) dénote le résultat booléen de l'application de la contrainte c sur la valuation θ

Le prédicat *meilleur* peut être défini de plusieurs façons différentes. Chaque définition associée à ce prédicat doit satisfaire deux propriétés, à savoir l'irréflexivité et la transitivité.

$\forall\theta\ \forall H \neg meilleur(\theta, \theta, H)$.

$\forall\theta, \eta, \omega\ \forall H\ meilleur(\theta, \eta, H) \wedge meilleur(\eta, \omega, H) \rightarrow meilleur(\theta, \omega, H)$.

Le prédicat *meilleur* doit respecte la sémantique de la hiérarchie : s'il existe une valuation θ dans S_0 qui satisfait complètement toutes les contraintes jusqu'au niveau k, alors toute valuation dans S doit satisfaire toutes les contraintes jusqu'au niveau k.

Si $\exists\theta \in S_0 \wedge \exists k>0$ /

$\forall i \in 1...k\ \forall c \in H_i\ c\,\theta = vrai$

alors $\forall\eta \in S\ \forall i \in 1...k\ \forall c \in H_i\ c\,\eta = vrai$.

Généralement, le prédicat *meilleur* ne produit pas un ordre total sur S_0. Il peut exister dans S deux valuations θ et η telles que θ n'est pas préférée à η et η n'est pas préférée à θ.

2.1.1 Les fonctions d'erreurs

Borning et son équipe dans [BFW92, BFW94] définissent plusieurs alternatives pour le comparateur *meilleur*. Dans chacune de ces alternatives, la fonction d'erreur $e(c\ \theta)$ est définie. Cette fonction retourne un nombre réel positif indiquant à quel degré la contrainte c n'est pas satisfaite par la valuation θ. Cette fonction retourne *0* si et seulement si la valuation θ satisfait la contrainte c. Pour n'importe quel domaine, la fonction d'erreur triviale qui retourne *0* si la contrainte est satisfaite et 1 sinon, peut être utilisée. Le comparateur utilisant cette fonction est appelé *comparateur-prédicat*. Pour un domaine qui est un espace métrique, il peut être souhaitable d'utiliser sa métrique pour déterminer l'erreur plutôt que d'utiliser la fonction d'erreur triviale définie précédemment. Par exemple, l'erreur pour la contrainte $x=y$ peut être la distance qui sépare x de y. Le comparateur utilisant ce type d'erreur est appelé *comparateur-métrique*.

La fonction d'erreur par niveau $E_i(H_i\ \theta)$ applique la fonction e sur chacune des contraintes dans $H_i = [c_1, ..., c_k]$. Le vecteur d'erreurs obtenu par cette application est : $E_i(H_i\ \theta) = [e(c_1\ \theta), ..., e(c_k\ \theta)]$. La séquence d'erreurs pour une hiérarchie de contraintes H contenant n niveaux est le vecteur : $[E_1(H_1\ \theta), ..., E_n(H_n\ \theta)]$.

2.1.2 Les Fonctions d'agrégation d'erreurs

Certains comparateurs auxquels nous nous sommes intéressés combinent les erreurs d'un niveau donné de la hiérarchie avant de comparer les valuations. Pour réaliser cela, ces comparateurs utilisent une fonction d'agrégation g. Cette fonction est appliquée au vecteur de valeurs réelles et retourne des valeurs qui sont comparées en utilisant deux relations: $<>_g$ et $<_g$. Quelques instances possibles de cette fonction d'agrégation g sont la somme des valeurs réelles dans le vecteur ou la valeur maximale dans le vecteur.

On exige que la relation $<_g$ soit irréflexive et transitive. On exige également que la relation $<>_g$ soit réflexive et symétrique. (On utilise la notation $<>_g$ au lieu d'utiliser = puisque pour quelques comparateurs la relation n'est pas transitive). La relation $<>_g$ indique deux valuations ne pouvant pas être ordonnées en utilisant la relation $<_g$ puisque, pour certains comparateurs, elles sont égales et que pour d'autres comparateurs elles sont incomparables.

La fonction d'agrégation G est une généralisation (entres niveaux) de la fonction g. Cette fonction est appliquée à une séquence d'erreurs et retourne une séquence de valeurs qui peuvent être comparées en utilisant les deux relations $<>_g$ et $<_g$.

Soit $R = [E(H_1\ \theta), \ldots, E(H_n\ \theta)]$ alors $G(R) = [g(E(H_1\ \theta)), \ldots, g(E(H_n\ \theta))]$.

L'ordre lexicographique $<_{lex}$ peut être défini d'une façon standard sur les séquences d'erreurs combinées $u_1,\ldots, u_n$ et $w_1,\ldots,w_n$.

$u_1,\ldots, u_n <_{lex} w_1,\ldots,w_n$ si $\exists k \in 1\ldots n$ tel que : $\forall i \in 1\ldots k\text{-}1 \quad u_i <>_g v_i \quad \wedge \quad u_k <_g v_k$

2.1.3 Types de comparateurs

Les séquences d'erreurs des contraintes dans les niveaux $H_1,\ldots, H_n$ sont comparées on utilisant l'ordre lexicographique. Si une valuation θ est considérée meilleure qu'une autre valuation η alors il existe un niveau k de la hiérarchie tel que pour $1 \leq i < k$, $g(E(H_i\ \theta)) <>_g g(E(H_i\ \eta))$, et au niveau k, $g(E(H_k\ \theta) <_g g(E(H_k\ \eta))$.

Borning dans [BFW92, BFW94] classifie deux types de comparateurs : les comparateurs locaux et globaux. Wilson dans [Wil92] distingue un troixième type de comparateurs appelé régionaux. Pour un comparateur local, chaque contrainte de la hiérarchie est considérée individuellement. Une valuation sera meilleure qu'une autre s'il existe un niveau dans lequel elle satisfait un sur-ensemble des contraintes de la seconde, et si pour tous les niveaux plus importants elles satisfont toutes les deux les mêmes contraintes. Pour un comparateur global, les erreurs de toutes les contraintes d'un niveau donné sont agrégées en utilisant la fonction g. Finalement, pour un comparateur régional, chaque contrainte d'un niveau donné est considérée individuellement (comme dans un comparateur local). Cependant, contrairement à un comparateur local, deux valuations qui sont incomparables aux niveaux importants de la hiérarchie, peuvent être comparées à un niveau moins important et par conséquent on obtient l'élimination d'une de ces deux valuations. En général, un comparateur global discrimine plus qu'un comparateur régional et ce dernier discrimine plus qu'un comparateur local. La suite de ce paragraphe présente les définitions formelles de ces comparateurs ainsi que quelques instances de ces définitions utilisées pour la résolution des hiérarchies de contraintes.

Comparateur local

On dira qu'une valuation θ est *Localement-Meilleur* qu'une autre valuation η, si pour chacune des contraintes jusqu'au niveau $k\text{-}1$, l'erreur obtenue sur θ est égale à celle obtenue sur η, et au niveau k, elle est strictement inférieure pour au moins une contrainte et inférieure ou égale pour tout le reste.

Formellement, cette définition est :

Localement-Meilleur$(\theta, \eta, H) \equiv \exists k \in 1\ldots n$ /

$\forall i \in 1\ldots k\text{-}1\ g(E(H_i\ \theta)) <>_g g(E(H_i\ \eta)) \wedge g(E(H_k\ \theta) <_g g(E(H_k\ \eta))$

avec $g(V) = V$ et $<>_g$ et $<_g$ sont définis par :

$$V <_g U \equiv \forall i\ v_i \leq u_i \wedge \exists j\ v_j < u_j .$$
$$V <>_g U \equiv \forall i\ v_i = u_i .$$

Si au niveau *k+1* de la hiérarchie, l'erreur obtenue sur θ est strictement supérieure à celle obtenue sur η pour une ou plusieurs contraintes, alors d'aprés la définition de ce comparateur, cela ne modifie en rien la véracité du comparateur *Localement-Meilleur.* La sémantique de la hiérarchie est toujours respectée.

Comparateur régional

On dira qu'une valuation θ est *Régionalement-Meilleur* qu'une autre valuation η, si pour tout niveau de la hiérarchie jusqu'au niveau *k-1* les deux valuations sont incomparables, et au niveau *k* (*k>1*), l'erreur obtenue après application de θ est strictement inférieure à celle obtenue après application de η pour au moins une contrainte, et inférieure ou égale pour tout le reste.

Formellement cette définition est :

Régionalement-Meilleur(θ, η, H) ≡ $\exists k \in 2...n$ /

$\forall i \in 1...k\text{-}1\ g(E(H_i\ \theta)) <>_g g(E(H_i\ \eta)) \wedge g(E(H_k\ \theta) <_g g(E(H_k\ \eta))$

avec $g(V) = V$ et $<>_g$ et $<_g$ sont définis comme suit :

$V <_g U \equiv \forall i\ v_i \leq u_i \wedge \exists j\ v_j < u_j .$

$V <>_g U \equiv \neg\ ((V <_g U) \vee (U <_g V))$.

Les définitions des comparateurs globaux que nous allons présenter considèrent que les contraintes de chaque niveau d'une hiérarchie peuvent être pondérées avec des poids réels. Pour les comparateurs locaux ou régionaux, cette pondération n'est pas utile puisque le résultat est équivalent avec ou sans pondération.

Comparateur Global

On dira qu'une valuation θ est *Globalement-Meilleur* qu'une autre valuation η, si pour chaque niveau de la hiérarchie jusqu'au niveau *k-1*, la valeur obtenue par agrégation des erreurs sur θ est égale à celle obtenue sur η, et au niveau *k*, elle est strictement inférieure.

Formellement, cette définition est :

Globalement-Meilleur(θ, η, *H*, *g*) ≡ $\exists k \in 1...n$ /

$\forall i \in 1...k\text{-}1\ g(E(H_i\ \theta)) <>_g g(E(H_i\ \eta)) \wedge g(E(H_k\ \theta) <_g g(E(H_k\ \eta))$.

Les relations $<>_g$ et $<_g$ sont équivalentes respectivement à = et < pour les réels, et *g(V)* est la fonction qui agrège la séquence d'erreurs dans le vecteur *V.*

Parmi les comparateurs globaux utilisant une fonction d'agrégation on peut citer par exemple : *Somme-Pondérée*, *Pire-Cas* ou encore *moindre-carrés*. Ces différents comparateurs prennent en compte des hiérarchies où les contraintes dans chaque niveau sont pondérées avec des poids réels. Les définitions de ces comparateurs sont:

Somme-Pondérée(θ, η, *H*) ≡ *Globalement-Meilleur*(θ, η, *H*, *g*)

avec $g(V) = \left(\sum_{i=1}^{|V|} w_i v_i \right)$.

Pire-Cas(θ, η, *H*) ≡ *Globalement-Meilleur*(θ, η, *H*, *g*)

avec $g(V) = (Max_{i:1..|V|} \{w_i v_i\})$.

Moindre-Carrés(θ, η, *H*) ≡ *Globalement-Meilleur*(θ, η, *H*, *g*)

avec $g(V) = \left(\sum_{i=1}^{|V|} w_i v_i^2 \right)$.

Indépendamment du choix de *Localement-Meilleur*, *Régionalement-Meilleur* ou d'une instance de *Globalement-Meilleur*, on peut choisir une fonction d'erreur appropriée pour les contraintes. Par exemple, le comparateur *Localement-Prédicat-Meilleur* est un dérivé de *Localement-Meilleur* (utilisant la fonction triviale qui retourne 0 si la contrainte est satisfaite et 1 sinon).

Un autre exemple est celui de *Localement-Métrique-Meilleur*, ce comparateur est un dérivé de *Localement-Meilleur* utilisant le domaine métrique pour compter l'erreur d'une contrainte. Nous avons aussi *Somme-Pondérée-Prédicat* et *Somme-Pondérée-Métrique* qui sont définis respectivement en utilisant l'erreur prédicat et l'erreur métrique.

Le comparateur *Nombre-Contraintes-Non-Satisfaites* est un cas particulier du comparateur *Somme-Pondérée-Prédicat* utilisant le poids 1 pour chacune des contraintes de chaque niveau de la hiérarchie. Ce comparateur compte le nombre de contraintes non satisfaites par niveau pour deux valuations quelconques, et établit leur comparaison.

On peut également remarquer l'existence du comparateur *Moindres-Carrés-Prédicat*, mais ce comparateur n'est pas particulièrement utile puisqu'il est équivalent au comparateur *Somme-Pondérée-Predicat* (puisque $1^2=1$). Le comparateur *Moindres-Carrés* est plutôt utilisé en se référant à l'erreur métrique, ce qui implique l'appellation suivante : *Moindres-Carrés-Métrique.*

2.1.4 Exemples illustratifs

Dans ce paragraphe, on présente deux exemples empruntés à [BFW94]. Le premier illustre la domination des contraintes situées au niveau fort d'une hiérarchie par rapport aux contraintes situées à un niveau plus faible. Le deuxième illustre les variétés des solutions obtenues par les différents comparateurs définis précédemment.

FIGURE 1 : Hiérarchie a trois niveaux de contraintes

niveau		*contraintes*
H_0	*requise*	*Celsius* × *1.8* = *Fahrenheit* − *32*
H_1	*forte*	*Fahrenheit* = 212
H_2	*faible*	*Celsius* = *0*

Le premier exemple considère la hiérarchie décrite dans la figure 1. Cette hiérarchie inclut la contrainte canonique Celsius-Fahrenheit. L'ensemble S_0 contient toutes les valuations qui satisfont la contrainte requise dans H_0. Pour cette hiérarchie, l'ensemble S_0 est infini, et contient tous les couples *(c,f)* de températures valide. S_0 = *{....,(-60, -76), (-40, -40), (0, 32), (10, 50), (100, 212), ...}* et S contient la seule valuation θ = *(100, 212)*. S ne contient aucune des autres valuations car celles-ci sont jugées noins bonnes que θ. Prenons par exemple la valuation η = *(10, 50)*, les valuations θ et η satisfont toutes les deux la contrainte requise dans H_0 et donc les erreurs de ces deux valuations sur ce niveau de la hiérarchie sont les mêmes. La valuation θ satisfait la contrainte du niveau H_1 de la hiérarchie tandis que la valuation η ne la satisfait pas, et donc il existe un niveau $k > 0$ ($k=1$) pour lequel l'erreur est strictement inférieure pour θ que celle obtenue pour η. Par conséquent, on a *Localement-Meilleur*(θ, η, H).

Considérons maintenant la valuation ω = *(0, 32)*. ω satisfait la contrainte dans H_1 et non celle dans H_2. Intuitivement, du fait que θ satisfait mieux les contraintes des niveaux supérieurs de la hiérarchie que ω, on a bien *Localement-Meilleur*(θ, ω, H). Cet exemple produit exactement le même ensemble S pour les comparateurs *Localement-Meilleur-Prédicat*, *Localement-Meilleur-Métrique* ou les instances du comparateur *Globalement-Meilleur*. Bien entendu, ceci ne peut pas être le cas général.

Le deuxième exemple illustre les variétés des solutions obtenues par les différents comparateurs présentés auparavant. Considérons un simple calculateur graphique comme il est décrit dans [BFW92]. Supposons que qu'on a la relation $A+B=C$. Initialement les valeurs des variables de cette relation sont respectivement : 2, 3 et 5. L'utilisateur désire modifier la valeur de la variable C et la rendre égale à 7. La hiérarchie dans la figure 2 exprime la sémantique de cette action. La contrainte $C = A+B$ est la contrainte requise et qui représente la relation de la somme entre les variables. La contrainte $C = 7$ est considérée au niveau le plus fort[1] de la hiérarchie (après le niveau des contraintes requises) puisqu'elle représente le désir de l'utilisateur. Les deux contraintes $A = 2$ et $B = 3$ sont considérées au dernier niveau de la hiérarchie et expriment le désir que le reste du système soit changé le moins possible en prenant en compte la nouvelle valeur de la variable C. Sans ces deux dernières, la valuation θ = *{ A = 1000000, B = –999993, C = 7}* peut parfaitement être une solution valide. On illustre maintenant les solutions des différents comparateurs.

FIGURE 2 : Hiérarchie d'un simple calculateur graphique.

niveau		*contraintes*
H_0	*requise*	$C = A + B$
H_1	*forte*	$C = 7$
H_2	*faible*	$A = 2$
	faible	$B = 3$

Le comparateur *Localement-Prédicat-Meilleur* produit deux valuations dans l'ensemble S :

1 Le fait de mettre cette contrainte au niveau des contraintes de plus grande préférence et non au niveau des contraintes requises, permet au système de ne pas modifier la variable C si cette dernière est déterminée par une contrainte requise.

$\theta = \{ A = 2, B = 5, C = 7 \}$ et $\eta = \{ A = 4, B = 3, C = 7 \}$.

Les séquences d'erreurs obtenues par θ et η sont respectivement :

$g(E(H_0\ \theta)) = g([0]) = [0]$, $g(E(H_1\ \theta)) = g([0]) = [0]$, $g(E(H_2\ \theta)) = g([0, 1]) = [0, 1]$ et $g(E(H_0\ \eta)) = g([0]) = [0]$, $g(E(H_1\ \eta)) = g([0]) = [0]$, $g(E(H_2\ \eta)) = g([1, 0]) = [1, 0]$.

Les deux valuations possèdent les mêmes séquences d'erreurs pour les deux niveaux 0 et 1 de la hiérarchie. Au dernier niveau de cette hiérarchie, la première valuation satisfait la contrainte $A = 2$ mais pas la contrainte $B = 3$ et la deuxième valuation satisfait la contrainte $B = 3$ mais pas la contrainte $A = 2$, ce qui fait que ces deux valuations ne satisfont pas les mêmes contraintes, et par conséquent elles sont mises toutes les deux dans l'ensemble S puisqu'elles ne sont pas comparables.

Le comparateur *Localement-Métrique-Meilleur* produit une infinité de solutions dans l'ensemble S :

$\{ A = x, B = 7-x, C = 7$ *pour* $x \in [2 \ldots 4] \}$. Aucune des valuations dans S n'est meilleure qu'une autre dans S. Par exemple, considérant les deux valuations suivantes dans S : la valuation $\theta = \{ A = 2.9, B = 4.1, C = 7 \}$ et $\eta = \{ A = 2, B = 5, C = 7 \}$. Les séquences d'erreurs obtenues par θ et η sont respectivement :

$g(E(H_0\ \theta)) = g([0.0]) = [0.0]$, $g(E(H_1\ \theta)) = g([0.0]) = [0.0]$, $g(E(H_2\ \theta)) = g([0.9, 1.1]) = [0.9, 1.1]$ et

$g(E(H_0\ \eta)) = g([0.0]) = [0.0]$, $g(E(H_1\ \eta)) = g([0.0]) = [0.0]$, $g(E(H_2\ \eta)) = g([0.0, 2.0]) = [0.0, 2.0]$.

Les deux valuations possèdent les mêmes séquences d'erreurs pour les deux niveaux 0 et 1 de la hiérarchie. Au dernier niveau de cette hiérarchie, la première valuation satisfait la contrainte $A = 2$ moins bien que la deuxième valuation (puisque $0.0 < 0.9$) et la deuxième valuation satisfait la contrainte $B = 3$ moins bien que la première valuation (puisque $1.1 < 2.0$), et par conséquent les deux valuations sont dans S. Cependant, les valuations du type $\{ A = 1000000, B = -999993, C = 7 \}$ sont considérées moins bonnes et ne sont pas dans S.

Le comparateur *Somme-Pondérée-Prédicat* produit le même ensemble de valuations que celui obtenu par le comparateur *Localement-Prédicat-Meilleur* si l'on considère que les deux contraintes du dernier niveau de la hiérarchie sont pondérées avec des poids équivalents. Dans le cas contraire, selon l'importance du poids, une des deux valuations sera choisie et mise dans S.

Le comparateur *Somme-Pondérée-Métrique* produit le même ensemble infini de valuations que celui obrenu par le comparateur *Localement-Métrique-Meilleur* si l'on considère que les deux contraintes du dernier niveau de la hiérarchie sont pondérées avec un poids équivalent. Dans le cas contraire, selon l'importance du poids, une des deux valuations suivantes : $\{A = 2, B = 5, C = 7\}$ $\{A = 4, B = 3, C = 7\}$, sera choisie et mise dans S.

Le comparateur *Moindres-Carrés-Métrique* produit la seule valuation : $\{A = 3, B = 4, C = 7\}$ lorsque le poids pondéré sur chacune des contraintes du dernier niveau de la hiérarchie est le même. Le comparateur *Pire-Cas-Métrique* produit la même valuation, en considérant la condition précédente. Les comparateurs régionaux produisent les mêmes solutions que les comparateurs locaux pour cette hiérarchie.

2.2 Comportement des comparateurs.

Une objection à ce qu'on a présenté serait : le fait d'avoir une hiérarchie contenant plus de deux niveaux (le premier pour les contraintes requises et le second pour les contraintes de préférences) n'est pas nécessaire puisque les contraintes de préférences peuvent être pondérées avec des poids pour déterminer l'ensemble *S*. Il y a deux raisons pour rejeter cette objection. La première est raison est conceptuelle et la deuxième est pragmatique. La raison conceptuelle constitut la raison fondamentale de l'utilisation des hiérarchies de contraintes. Ces raisons sont :

- ne pas laisser les contraintes contraintes faibles influencer le résultat. Pour illustrer cette raison conceptuelle, considérons une application graphique interactive où nous avons à déplacer une ligne par la souris. La ligne est contrainte par trois contraintes : deux contraintes sont considérées au niveau fort de la hiérarchie et la troisième contrainte est considérée à un niveau en dessous. Les deux premières contraintes sont : " *la ligne doit rester horizontale*", " *un point extrémité de la ligne doit suivre le déplacement de la souris*". La troisième contrainte est : "*la ligne est attachée à des points fixes dans le diagramme*". Le désir de l'utilisateur dans ce cas est que la ligne demeure exactement horizontale tout en suivant précisément la souris (laissant la contrainte faible non satisfaite), au lieu de laisser la contrainte faible influencer le résultat (ce qui résulterait en ce que la ligne serait proche de l'horizontale ou en ce qu'elle serait proche de la position de la souris).
- les solutions d'une hiérarchie contenant plusieurs niveaux de préférence (où un niveau supérieur domine complètement un niveau inférieur) peuvent être obtenues beaucoup plus facilement que si la hiérarchie possède un seul niveau de préférence contenant les contraintes pondérées (bien sûr ceci dépend des algorithmes utilisés).

La plupart des concepts des hiérarchies de contraintes sont dérivés des concepts de certaines branches de la recherche opérationnelle comme la programmation linéaire [Mur83], la programmation par buts [Ign83] ou encore la programmation par buts généralisée [Ign85]. Le domaine des contraintes dans la recherche opérationnelle est souvent l'ensemble des réels *R*, et parfois le domaine est celui des entiers naturels *N*, selon le type de problème.

Dans [BFW92, Wil92], l'approche de la hiérarchie de contraintes a été précédée par l'approche des problèmes multi-objectifs qui consiste à considérer la fonction objectif en priorité. Le concept des solutions obtenues par le comparateur *Localement-Meilleur* est dérivé de celui du vecteur minimal d'un problème de programmation linéaire multi-objectifs. D'une façon identique, le concept des solutions obtenues par le comparateur *Somme-Pondérée* ou par le comparateur *Pire-Cas* sont tous les deux dérivés de concepts analogues des problèmes de programmation linéaire multi-objectifs et programmation par buts généralisée.

Pour une fonction d'erreur donnée, on peut remarquer l'existence de plusieurs relations entre les comparateurs locaux et les comparateurs globaux :

Proposition 2.1 : $\forall\theta, \eta\ \forall H$ *Localement-Métrique-Meilleur*$(\theta, \eta, H) \rightarrow$ *Somme-Pondérée-Métrique*(θ, η, H).

Preuve : Supposons que *Localement-Métrique-Meilleur*(θ, η, H) est vrai, et donc qu'il existe un niveau $k>0$ dans H tel que l'erreur résultante après l'application à chacune des contraintes sur θ jusqu'au niveau $k-1$ est égale à celle obtenue après application sur η. Ceci implique bien que la somme des erreurs pondérées par les poids après application sur θ de toutes les contraintes jusqu'au niveau $k-1$ est égale à la somme des erreurs pondérées par les poids après application de η. De plus au niveau k, et par utilisation de *Localement-Métrique-Meilleur*(θ, η, H), l'erreur après application

sur θ est strictement inférieure pour au moins une contrainte et inférieure ou égale pour toutes les contraintes restantes de ce niveau après application sur η. Ceci implique bien aussi que la somme des erreurs pondérées par les poids après application sur θ des contraintes du niveau k est strictement inférieure à la somme des erreurs pondérées par les poids après application sur η. Par conséquent, on a bien *Somme-Pondérée-Métrique*(θ, η, *H*).

Corollaire 2.1: Pour une hiérarchie de contraintes donnée, soit S_{LMM} l'ensemble des solutions obtenu par l'utilisation du comparateur *Localement-Métrique-Meilleur*, et S_{SPM} celui obtenu par l'utilisation du comparateur *Somme-Pondérée*, on a : $S_{SPM} \subseteq S_{LMM}$.

Proposition 2.2 : ∀θ, η ∀*H Localement-Métrique-Meilleur*(θ, η, *H*) → *Moindre-Carré*(θ, η, *H*).

Preuve : Identique à celle de la proposition 1.

Corollaire 2.2: Pour une hiérarchie de contraintes donnée, soit S_{LMM} l'ensemble des solutions obtenu par l'utilisation du comparateur *Localement-Métrique-Meilleur*, et S_{MC} celui obtenu par l'utilisation du comparateur *Moindre-Carré*, on a : $S_{MC} \subseteq S_{LMM}$.

Les propositions 2.1 et 2.2 concernent particulièrement deux instances du schéma du comparateur *Globalement-Meilleur*. Pour une fonction d'agrégation g arbitraire, le comparateur *Localement-Meilleur* n'implique pas toutes les instances du comparateur *Globalement-Meilleur*. En particulier, *Localement-Meilleur* n'implique pas *Pire-Cas* ou encore d'autres comparateurs que nous définirons dans les chapitres suivant.

2.3 Erreurs produites par des inégalités

Certains problèmes sont soulevés lorsqu'on a à résoudre des contraintes d'inégalité stricte en utilisant l'erreur métrique. Par exemple: que doit être la fonction d'erreur pour la contrainte $x>y$, lorsque le domaine des variables x et y est l'ensemble des réels ? Si x est supérieur à y alors l'erreur doit être nulle. Si x n'est pas supérieur à y, on aimerait que l'erreur soit égale au plus petit réel proche de y-x. L'évidente fonction d'erreur $e(x>y) = 0$ si $x>y$ sinon y-x. n'est pas correcte puisqu'elle retourne la valeur 0 lorsque la valeur de x est égale à celle de y. Cependant, si l'erreur est égale a un nombre d positif lorsque $x=y$, alors on aura une erreur plus petite lorsque $y=x+d/2$, et donc ceci contredit notre souhait que l'erreur diminue quand la valeur de x s'approche de celle de y. Pour résoudre ce problème, un nombre infiniment petit noté ε est introduit [Rob66] : ε est plus grand que 0 et plus petit que tout nombre positif. En utilisant ε, la fonction erreur e est définie comme suite :

$$e(x>y) = \begin{cases} y - x \text{ si } x<y \\ \varepsilon \text{ si } x=y \\ 0 \text{ si } x>y \end{cases}$$

$$e(x \neq y) = \begin{cases} 0 \text{ si } y \neq x \\ \varepsilon \text{ si } x=y \end{cases}$$

$$e(x<y) = \begin{cases} x - y \text{ si } x>y \\ \varepsilon \text{ si } x=y \\ 0 \text{ si } x<y \end{cases}$$

Il est à noter que ε est ajouté seulement aux valeurs possibles de la fonction erreur et non au domaine D. Si l'on essaye de changer le domaine lui même, on tombera dans le problème cité auparavant[1].

2.4 Existence de solutions

Intuitivement, on peut supposer que l'ensemble S des solutions à une hiérarchie de contraintes est non vide lorsque l'ensemble S_0 de solutions pour les contraintes requises de cette hiérarchie est non vide. Cependant, ceci n'est pas toujours le cas. Considérons l'erreur métrique et la hiérarchie suivante: *requise* $x>0$ et *fort* $x=0$ avec x ayant pour domaine R. L'ensemble S_0 est celui de R^{+*} mais l'ensemble S est vide puisque pour chaque valuation dans l'ensemble S_0 (par exemple d), il existe une autre valuation (par exemple $d/2$) qui satisfait mieux la contrainte de préférence $x=0$.

Proposition 2.3: Si S_0 est non vide et fini alors S est non vide.

Preuve : Supposons que S soit vide, et prenons une valuation θ_1 de S_0. Puisque $\theta_1 \notin S$, il doit exister une autre valuation $\theta_2 \in S_0$ telle qu'on ait *meilleur*(θ_2, θ_1, H). D'une façon similaire, puisque $\theta_2 \notin S$, il doit exister une autre valuation $\theta_3 \in S_0$ telle qu'on ait *meilleur*(θ_3, θ_2, H), et ainsi de suite pour la chaîne infinie θ_4, θ_5... θ_n. Puisque *meilleur* est transitive on a par induction que $\forall i,j > 0$ [$i > j \rightarrow$ *meilleur*(θ_i,θ_j, H)]. La propriété d'irréflexivité du comparateur *meilleur* impose que $\forall i > 0$ on a $\neg$ *meilleur*(θ_i,θ_i, H) et donc toutes les θ_i sont distinctes. On aboutit donc à leur infinité, ce qui est en contradiction avec l'hypothèse.

La plupart des applications pratiques utilisent des hiérarchies de contraintes finies. Par exemple pour les programmes en programmation impérative par contraintes ou encore en programmation logique avec hiérarchie de contraintes, si le programme se termine, l'ensemble des contraintes résultant doit être fini. La proposition suivante nous montre que dans plusieurs cas d'importance pratique, si les contraintes requises peuvent être satisfaites, alors une solution à la hiérarchie existe.

Proposition 2.4: Si S_0 est non vide, et H est fini et le comparateur utilisé utilise l'erreur prédicat, alors S est non vide.

Preuve : Supposons que S soit vide, et utilisons les mêmes arguments décrits dans la preuve précédente, on présume donc qu'il doit exister un nombre infini de valuations distinctes $\theta_i \in S_0$. Cependant, si le comparateur est un prédicat, une valuation ne peut être meilleure qu'une autre si toutes les deux satisfont exactement le même sous ensemble de contraintes dans H. Par conséquent, chacune des valuations $\theta_i \in S_0$ doit satisfaire un sous ensemble de contraintes différent dans H, et donc ceci suppose que l'on doit avoir H infini ce qui est en contradiction avec l'hypothèse.

2.5 Les aspects non-monotones des comparateurs

Il est temps de considérer les effets de l'ajout d'une contrainte à une hiérarchie existante pour raffiner l'ensemble de valuations solutions de la hiérarchie [Wil92, Bor81, JM87]. Ceci peut avoir lieu, par exemple dans les environnements graphiques interactifs où l'on possède une série de buts successifs relatifs à la position d'un objet sur l'écran. Chaque but représente une commande de l'uti-

1. Que doit être l'erreur pour la contrainte 0>ε/2 ?. Selon la définition, l'erreur est de ε/2, mais cette erreur est inférieure à celle de la contrainte 0>0, quoique la contrainte 0>0 soit mieux satisfaite.

lisateur. Si une des commandes n'a pas encore été exécutée, et si l'on veut déterminer la position courante de l'objet pour pouvoir l'afficher, on aura besoin de manipuler les contraintes d'une façon incrémentale.

Malheureusement, les comparateurs qui respectent la hiérarchie ne possèdent pas la propriété d'ordre décrite ci-après. Intuitivement, ceci veut dire que l'ajout d'une nouvelle contrainte à la hiérarchie peut conduire à une solution qui n'est pas une solution construite à partir de la solution précédente, mais plutôt une solution différente.

Définition 2.2

Soient H et J deux hiérarchies de contraintes. Soit C un comparateur. C est dit ordonné si et seulement si : $S_{(H\cup J)}(C) \subseteq S_{(H)}(C)$. Un comparateur qui n'est pas ordonné est appelé désordonné [WB89].

Il est à noter qu'en programmation logique avec contraintes on a l'aspect ordonné, puisque l'ajout d'une contrainte requise à un ensemble initial de contraintes requises ne peut que réduire ou laisser constant l'ensemble de valuations qui satisfont cet ensemble initial de contraintes. Cette propriété est similaire à la propriété de "*la stabilité de rejet*" décrite dans [Wyk80].

Proposition 2.5 : Soit D un domaine contenant plus qu'un élément. Tout comparateur C qui respecte la hiérarchie est désordonné.

Preuve : Soit H=*{faible x=a}* et soit J=*{forte x=b}*, avec a et b deux éléments distincts du domaine. Soit C un comparateur qui respecte la hiérarchie. $S_{(H)}(C)$ contient la valuation qui associe x à la valeur a, et $S_{(H\cup J)}(C)$ contient uniquement la valuation qui associe x à la valeur b. On n'a pas la relation $S_{(H\cup J)}(C) \subseteq S_{(H)}(C)$ puisque a et b sont distincts. Ceci implique bien que C est non ordonné.

Donc, si l'on possède un résolveur incrémental, l'ajout d'une nouvelle contrainte doit en général nous obliger à rétracter la solution précédente. L'aspect désordonné des comparateurs montre que l'implémentation d'un résolveur incrémental général présente quelques difficultés. Cependant, un certain nombre de solutions sont disponibles pour des cas pratiques. Considérons par exemple une application graphique interactive. Comme il est mentionné plus haut, une implémentation raisonnable en programmation logique peut être de créer une série de buts en relation avec les différents états successifs d'un objet en mouvement à l'écran. En principe, on peut déterminer une solution d'une façon incrémentale pour un de ces états, mais après l'ajout d'une nouvelle contrainte sur cet état, la précédente solution peut devenir invalide.

En pratique et pour les interfaces graphiques, la programmation logique avec hiérarchie de contraintes est utilisée dans le sens où toutes les contraintes sur un état donné doivent être connues à l'avance [WB89]. De plus, les contraintes doivent porter seulement sur les états futurs et non sur les états courants. Une solution à ce problème serait d'inclure une variante d'opérateurs dans le langage. Un opérateur doit agir comme une coupure et doit aussi résoudre la hiérarchie courante et enfin imposer que la solution trouvée soit une contrainte requise sur les variables de la hiérarchie. Après ce processus, l'ajout d'une nouvelle contrainte au système ne peut que rendre la solution plus fine et non l'invalider.

Synthèse du chapitre

Nous avons présenté dans ce chapitre l'aspect formel d'une hiérarchie de contraintes. La résolution d'une hiérarchie passe d'abord par la résolution des contraintes dures dans cette hiérarchie. L'ensemble des valuations obtenu à l'issue de cette résolution est ensuite utilisé pour bâtir l'ensemble des valuations qui sont solutions de la hiérarchie. La construction de ce dernier ensemble nécessite l'utilisation d'un critère de comparaison. Ce critère de comparaison peut être local, régional ou global.

Nous avons présenté des instances de ces trois types de comparateurs et montré leurs effets sur des exemples de hiérarchies de contraintes. Nous avons présenté également le comportement de ces types de comparateurs, certaines de leurs propriétés et les relations existantes entres leurs différentes instances. Enfin, nous avons montré le comportement de ces types de comparateurs dans un environnement incrémental où les contraintes sont ajoutées (ou retirées) à la hiérarchie.

3 Extensions de la théorie des hiérarchies de contraintes

Les cinq parties qui composent ce chapitre visent à donner un aperçu sur les extensions de la théorie des hiérarchies de contraintes décrite au chapitre précédent. Dans un premier temps, on verra les annotations que l'on peut porter sur des variables dans une hiérarchie de contraintes. Nous montrons l'intérêt de ces annotations à travers des exemples pratiques et réels. Dans un deuxième temps, nous verrons la notion des hiérarchies partiellement ordonnées et ce qu'il résulte de cet ordre. En troisième partie de ce chapitre, nous présenterons l'intégration des fonctions objectifs aux hiérarchies de contraintes. En quatrième partie, on discutera de la variation du critère selon les niveaux d'une hiérarchie ainsi que de l'intérêt de cette variation. Et en dernière partie, nous parlerons de l'intégration des hiérarchies de contraintes dans la programmation logique par contraintes.

3.1 Les annotations *lecture-seulement* et *écriture-seulement*

3.1.1 L'annotation *lecture-seulement*

Les systèmes de contraintes basés sur le modèle de perturbation utilisent souvent le principe de l'annotation *lecture-seulement*. Ceci a pour but d'aider à limiter le choix des variables à recalculer pour resatisfaire les contraintes après une perturbation introduite dans le système. Les hiérarchies de contraintes peuvent être vues comme une solution alternative pour spécifier ce choix sans avoir à soulever la généralité des contraintes multi-directionnelles. Cependant, la notion de *lecture-seulement* est utile dans les hiérarchies de contraintes multi-directionnelles. Cette notion est utile à la résolution de contraintes qui font référence à un support externe pour l'acquisition d'une donnée ou qui font référence à une autre source d'information hors site. Un exemple simple est celui de la souris. La contrainte possédant un point qui suit la position de la souris doit être en *lecture-seulement* sur cette position.

Un autre exemple est celui des contraintes qui sont des relations entre des états anciens et des états nouveaux d'un robot. Dans ce cas, on aimerait que les états anciens soient en *lecture-seulement* afin que le futur ne change pas le passé.

Intuitivement, lorsqu'on choisit les meilleures solutions d'une hiérarchie de contraintes, les contraintes ne doivent pas influencer le choix des valeurs de leurs variables marquées par l'annotation *lecture-seulement*. Au contraire, les contraintes peuvent influencer le choix des valeurs de leurs variables qui ne sont pas annotées. Cependant, on veut que les contraintes soient satisfaites si possible tout en respectant leurs niveaux de préférence. En particulier, lorsque les contraintes requises sont soumises à l'annotation *lecture-seulement*, on doit avoir leur satisfaction. Une esquisse informelle de la définition est donnée dans [BFW92, BFW94] et consiste à remplacer les occurrences des variables (marquées par la notation *lecture-seulement*) par des constantes. Ceci a pour but d'empêcher la contrainte d'influencer le choix de ces variables marquées. Ainsi nous commençons la définition de l'ensemble de solutions de la hiérarchie H en formulant un ensemble Q de hiérarchies de contraintes. Chaque élément dans Q est une hiérarchie de contraintes avec un domaine arbitraire d'éléments substitué pour les variables marquées par la notation *lecture-seulement* .

Cette démarche est réalisée en suivant les étapes suivantes: une valeur pour la variable est choisie et une hiérarchie est formulée en considérant cette valeur. Après avoir choisi toutes les valeurs possibles, les solutions résultantes sont triées en enlevant celles qui sont incorrectes (il est à signaler que cette approche a pour objectif de spécifier la sémantique de la notation *lecture-seulement*, et que cela ne construit pas un algorithme raisonnable pour la résolution d'une hiérarchie de contraintes. Les algorithmes de résolution sont discutés dans les chapitres suivants de ce mémoire).

3.1.2 Exemple illustratif

La figure 3 montre une hiérarchie à deux niveaux de préférences (cet exemple est emprunté à [BFW94]). Le symbole *"?"* symbolise la notation *lecture-seulement*. Les éléments de Q sont formulés en remplaçant les occurrences de la variable y par les éléments du domaine D.

FIGURE 3 : Hiérarchie contenant l'annotation lecture-seulement

Hiérarchie initiale	$q \in Q$ *formulé en remplaçant* $y?$ *par un* $d \in D$			
requise $x = y?$	$y? \rightarrow -3.2$ $x = -3.2$	$y? \rightarrow 4$ $x = 4$	$y? \rightarrow 7.8$ $x = 7.8$	....
forte $x = 3$	$x = 3$	$x = 3$	$x = 3$	....
faible $y = 4$	$y = 4$	$y = 4$	$y = 4$	....

Dans la figure 4, on résout les hiérarchies de contraintes dans Q en éliminant les valuations qui associent aux occurrences des variables (non marquées) une valeur différente de celle associée à l'occurrence marquée par la notation *lecture-seulement*. On observe que la valuation $\{y \rightarrow 4,\ x \rightarrow 4\}$ est la seule consistante et constitue une solution à la hiérarchie initiale.

FIGURE 4 : Hiérarchie contenant la notation lecture-seulement

	q ∈ *Q formulé en remplaçant y? par un d* ∈ *D*	
valuation ϕ	*y?* → *−3.2*	*y?* → *4*
hiérarchie q	*requise x = −3.2* *forte x = 3* *faible y = 4*	*requise x = 4* *forte x = 3* *faible y = 4*
valuation θ	*{y* → *4, x* → *−3.2 }*	*{y* → *4, x* → *4}*
consistance	*y?*ϕ ≠ y θ	*y?*ϕ = y θ
jugement	*retirée*	*gardée*

3.1.3 Définitions formelles de la notion *lecture-seulement*

Les définitions formelles illustrant la sémantique de cette notion de *lecture-seulement* sont les suivantes :

Définition 3.1:

Soit θ une valuation, soit l'ensemble des variables $v_1, \ldots, v_n$ les variables de θ.

Soit la valuation θ' telle que θ' = {θ omet $w_1, \ldots, w_n$ }.
Le domaine de la valuation θ' est $\{v_1, \ldots, v_n\} - \{w_1, \ldots, w_n\}$ et
pour tout v appartenant à ce domaine on a : θ'v=θv.
D'une façon similaire, si S est un ensemble de valuations,
S omet $w_1, \ldots, w_n \equiv \{\theta$ omet $w_1, \ldots, w_n / \theta \in S\}$.

Définition 3.2 :

Soit H une hiérarchie de contraintes, et soit D le domaine des contraintes de cette hiérarchie. Soit $v_1, \ldots, v_n$ l'ensemble des variables de H. H a une ou plusieurs occurrences marquées par *lecture-seulement*. Soient $w_1,\ldots,w_m$ des nouvelles variables non référencées dans H et soit J la hiérarchie résultante suite à la substitution de w_i aux occurrences de la variable v_i marquée (les occurrences de la variable v_j non marquées ne sont pas concernées). On définit Q par l'ensemble de toutes les hiérarchies $J\,\rho$ où chaque ρ est formé en substituant à w_i un élément arbitraire du domaine D.

$$Q = \{J\,\rho / d_1 \in D, \ldots, d_1 \in D\} \text{ et } \rho = \{ w_1 \rightarrow d_1, \ldots, w_m \rightarrow d_m\}.$$

Soit *solutions*$(J\,\rho)$ l'ensemble des solutions de $J\,\rho$ (ici, on utilise la définition de l'ensemble de solutions S présenté au chapitre précédent puisque J ne possède pas maintenant de variables marquées par la notation *lecture-seulement*). L'ensemble de solutions consistantes de $J\,\rho$ est défini par :

$$Consistante(J\,\rho) = \{\theta / \theta \in solutions(J\,\rho) \wedge (w_1\,\rho = v_1\,\theta) \wedge \ldots \wedge (w_m\,\rho = v_n\,\theta)\}.$$

Une solution est consistante lorsque ρ associe un élément d_i du domaine à w_i et que θ associe le même élément d_i à v_i. L'ensemble désiré de solutions à la hiérarchie H est l'ensemble des solutions consistantes omettant les valeurs associées aux nouvelles variables w_i introduites.

$$Solutions(H) = (\cup_{(J\,\rho) \in Q} Consistante(J\,\rho)) \text{ omet } w_1,\ldots,w_n .$$

Proposition 3.1 : Pour une hiérarchie H ne contenant que des contraintes requises, soit H' la même hiérarchie que H mais sans la notation *lecture-seulement*, alors $Solutions(H) = Solutions(H')$.

Preuve 3.1 : Dans un premier temps on prouve que : $Solutions(H) \supseteq Solution(H')$. Soit $\{v_1, \ldots, v_n\}$, $\{w_1, \ldots, w_n\}$ et J défini comme auparavant. Soit θ une solution pour H'. On définit $\rho = \{ w_i \rightarrow v_i\,\theta, \ldots, w_m \rightarrow v_m\,\theta \}$ (en d'autres termes: si θ associe d_i à v_i, alors ρ associe d_i à la variable w_i correspondante) et donc il est clair que $\theta \in Solutions(J\ \rho)$ et θ est constante. Par conséquent, $\theta \in Solutions(H)$. Dans un second temps on prouve que : $Solutions(H) \subseteq Solution(H')$. Soit θ une solution pour H. Par définition θ est une solution consistante de $(J\ \rho)$ pour un ρ donné. Puisque H contient seulement des contraintes requises et θ est consistante avec ρ, θ satisfait aussi toutes les contraintes de H' et donc $\theta \in Solutions(H')$.

3.1.4 Exemples pratiques illustrant l'utilisation de l'annotation *lecture-seulement.*

Dans les langages de programmation logique, la notation lecture-seulement a été initialement introduite dans Concurrent Prolog [Sha86, Sar85, Mah87] pour un propos différent. Elle a été introduite pour le contrôle de communication et synchronisation dans des réseaux de processus.

Un exemple trivial mais beaucoup utilisé est une contrainte de la forme *A?+B?+C?=somme*. En peut voir qu'ici l'annotation *lecture-seulement* empêche l'utilisateur d'attribuer une valeur à la variable *somme* et d'avoir à propager ce changement à *A*, *B* ou *C*, mais permet à l'utilisateur d'attribuer une valeur à *A*, *B* ou *C*.

Comme il est mentionné au paragraphe 3.1.1, une utilisation importante de cette annotation est concrétisée dans des contraintes qui font référence à un support extérieur ou une source d'information située à l'extérieur du système. Par exemple si l'on a la contrainte : *le point P doit suivre le déplacement de la souris*. La contrainte doit être marquée par l'annotation *lecture-seulement* sur la position de la souris. *P = position.souris?*.

Considérons un autre exemple, supposons que l'on dispose d'un affichage avec une simple barre de défilement sur l'écran. Lorsque l'ascenseur se déplace entre le point haut et le point bas de la barre de défilement, on aimerait que les points haut et bas de la barre restent à leurs positions. Cependant, on aimerait être capable de repositionner la barre de défilement et donc le fait de fixer les points haut et bas de la barre ne constitue pas une solution correcte (on peut presque réaliser le résultat désiré en étiquetant par *fort* (mais non *requise*) des contraintes de fixation ou maintien des points haut et bas de la souris. Le problème est que d'autres contraintes sur les variables contenant les valeurs de position du curseur peuvent êtres étiquetées par *très-fort*, et par conséquent le point haut ou bas de la barre de défilement peut être changé). Pour résoudre ce problème efficacement, on définit une contrainte dans la figure 5. Cette contrainte est relative aux positions de l'ascenseur, des points haut et bas de la barre de défilement marqués par l'annotation *lecture-seulement* et d'une variable de pourcentage.

Il est bien évident que l'annotation sur les deux points haut et bas est spécifique à cette contrainte et donc la barre de défilement peut être repositionnée par une autre contrainte de déplacement.

FIGURE 5 : Contrainte de la barre de défilement relative à une position.

$$pourcentage = \frac{ascenseur - bas?}{haut? - bas?}$$

3.1.5 Circularité

Si pour plusieurs hiérarchies l'ensemble de solutions est intuitivement clair, cette clarté s'évanouit souvent lorsque la hiérarchie contient des contraintes qui forment un cycle. On présente deux exemples pour montrer ce phénomène [Wil92]. Il s'agit de cas pathologiques qui ne devraient pas être rencontrés dans les applications réelles, mais la théorie doit spécifier comment ils doivent être traitées.

Les deux hiérarchies de la figure 6 contiennent chacune un cycle entre leurs variables marquées.

En H_1, aucune contrainte dans le cycle n'est plus restrictive qu'une autre et donc l'information peut circuler et rendre une solution. Pour cette hiérarchie l'ensemble de solutions est l'ensemble infini $\{\{x \rightarrow d+1, y \rightarrow d\} / d \in R\}$.

En H_2, la contrainte $x? = y+1$ est plus restrictive que la contrainte $x \geq y?$. Ainsi l'information d'inégalité transite. Pour cette hiérarchie, l'ensemble de solutions est $\{\{x \rightarrow 21, y \rightarrow 20\}\}$.

FIGURE 6 : Hierarchies contenant des contraintes qui forment des cycles

$H_1 = \{$*requise* $x? = y+1$, *requise* $x = y?+1\}$ et

$H_2 = \{$*requise* $x? = y+1$, *requise* $y = 20$, *requise* $x \geq y?\}$.

3.1.6 L'annotation *écriture-seulement*

Outre l'annotation *lecture-seulement*, il convient aussi de définir la notation *écriture-seulement*. Lorsqu'une variable est marquée par cette notation, ceci veut dire que l'on veut que l'information circule de la contrainte vers la variable et non le contraire. On pourrait définir les effets de cette notation d'une manière identique à celle de la notation *lecture-seulement*. Cependant, il est plus facile de définir la notation *écriture-seulement* en terme de celle de *lecture-seulement*.

Définition 3.4 :

Soit H une hiérarchie de contraintes, et soit D le domaine des contraintes de cette hiérarchie. Soit $v_1, \ldots, v_n$ l'ensemble des variables de H. H a une ou plusieurs occurrences marquées par la notation *écriture-seulement*. Soient $w_1, \ldots, w_m$ des nouvelles variables non référencées dans H et soit J la hiérarchie résultante suite à la substitution de w_i par les occurrences de la variable v_i marquée (les occurrences de la variable v_i non marquées ne sont pas concernées). Soit J' la hiérarchie formée par ajout des contraintes *fortes* $v_i = w_i?$ pour $1 \leq i \leq m$ dans J. L'ensemble des solutions désiré de H est celui de J' en omettant les w_i.

Solutions(J) = *Solutions(J')* *omet* $w_1, \ldots, w_m$. La définition de *Solutions(J')* est celle donnée précédement.

Par exemple, soit H= *{ requise x! = y, forte x = 4, faible y = 3 }*. La contrainte $x = 4$ est étiquetée par *forte*, ce qui fait d'elle une contrainte plus importante à satisfaire que celle étiquetée par *faible*. Les informations ne sont autorisées à circuler que de la variable y vers la variable x dans la contrainte $x! = y$ puisque x est marquée par la notation *écriture-seulement*. En utilisant la définition, la hiérarchie J' est formée en remplaçant $x!$ par une nouvelle variable introduite w et en ajoutant la contrainte requise $x = w?$. Soit J' = { *requise* $w = y$, *requise* $x = w?$, *forte* $x = 4$, *faible* $y = 3$ }. L'ensemble de solutions à J' est $\{\{w \rightarrow 3, x \rightarrow 3, y \rightarrow 3\}\}$. L'ensemble de solutions à la hiérarchie H est celui de J' en omettant la nouvelle variable introduite, soit $Solutions(H) = \{\{x \rightarrow 3, y \rightarrow 3\}\}$.

3.2 Hiérarchies partiellement ordonnées

Dans certaines applications, le fait d'imposer un ordre total sur les étiquettes des contraintes (ordre total sur les niveaux de la hiérarchie) peut être vu comme une sur-spécification du problème. Pour remédier à ce problème, l'ensemble des solutions d'une hiérarchie à ordre partiel sur ces niveaux a été défini. Pour une hiérarchie partiellement ordonnée, l'ordre des étiquettes des contraintes non requises est partiel.

Informellement, l'ensemble de solutions d'une hiérarchie partiellement ordonnée est défini par la formation de l'ensemble de toutes les hiérarchies à ordre total et qui soient consistantes avec la hiérarchie initiale. Ces hiérarchies à ordre total sont obtenues par l'ajout des ordres permissibles entre l'ordre partiel sur les niveaux de la hiérarchie initiale : *inférieur à* , *supérieur à* ou *égal à*. L'ensemble des solutions désiré est l'union des ensembles de solutions de ces hiérarchies.

Définition 3.5 :

Si H est une hiérarchie à ordre partiel sur ces niveaux, la hiérarchie H' à ordre total sur ces niveaux est construite si :

Les deux hiérarchies H et H' contiennent les mêmes contraintes

Il existe une correspondance m des étiquettes de H à celles de H' telle que

si $s_i < s_j$ dans H alors $m(s_i) < m(s_j)$ dans H'

$\forall i\ s_i\ c_i \in H\ s_i$ et seulement si $m(s_i)\ c_i \in H'$.

Définition 3.6 :

Soit H une hiérarchie à ordre partiel sur ces niveaux alors :

$Solutions(H) = (\cup_{H' \in \mathrm{H}} Solutions(H'))$ où est H l'ensemble de toutes les hiérarchies à ordre total sur leurs niveaux et constantes avec H.

Comme exemple trivial, considérons la hiérarchie suivante : H = *{ petit x=3, faible x=4 }*. Les étiquettes *petit* et *faible* sont toutes les deux non requises et il n'y a pas d'ordre spécifié entre elles. Les ordres totaux qui sont consistants avec cet ordre partiel sont : *petit* est *supérieur à faible*, *petit* est *inférieur à faible* et finalement *petit* est *égal à faible*. En utilisant le comparateur local *localement-prédicat-meilleur* les ensembles de solutions de ces hiérarchies sont respectivement : $\{\{x \rightarrow 3\}\}$, $\{\{x \rightarrow 4\}\}$ et $\{\{x \rightarrow 3\}, \{x \rightarrow 4\}\}$ et donc l'ensemble de solutions à la hiérarchie H est : $\{\{x \rightarrow 3\}, \{x \rightarrow 4\}\}$.

La définition 3.6 entraîne l'ajout de tous les ordres entre les étiquettes de la hiérarchie. Pour les comparateurs locaux, l'ordre *égal à* est non nécessaire puisque chaque solution à la hiérarchie d'ordre total *égal à* est aussi une solution à l'une des hiérarchies à ordre total dans *{inférieur à , supérieur à}*. Ceci n'est pas le cas pour les comparateurs globaux. Considérons par exemple le comparateur *Moindre-Carré*, dans l'exemple précédent les solutions des hiérarchies à ordre total sont respectivement $\{\{x \rightarrow 3\}\}$, $\{\{x \rightarrow 4\}\}$ et $\{\{x \rightarrow 3.5\}\}$ et donc l'ensemble de solutions à la hiérarchie H est : $\{\{x \rightarrow 3\}, \{x \rightarrow 4\}, \{x \rightarrow 3.5\}\}$.

Borning et son équipe dans [BMM+89] considèrent une variante de la définition des solutions aux hiérarchies à ordre partiel. Non seulement deux étiquettes à ordre partiel sont combinées en une seule (i.e *égal à*) mais aussi tous les poids possibles entre les contraintes doivent être utilisés. Dans l'exemple précédent et en utilisant le comparateur global *Moindre-Carré*, l'ensemble infini de hiérarchies à ordre total est : *{{fort x=3, très-faible x=4}, {très-faible x=3, fort x=4}, {moyen[w_1] x=3, moyen[w_2] x=4} pour tout nombre positif w1 et w2}*. L'ensemble de solutions dans ce cas est $\{\{x \rightarrow a\} / a \in [3..4]\}$.

3.3 Les fonctions objectifs

L'objectif dans certains problèmes standards basés sur la programmation linéaire est de minimiser (ou maximiser) la valeur de la fonction linéaire $z(x_1, ..., x_k) = a_1 x_1 + ... + a_k x_k$ comportant k variables de domaine de définition R^{+*} [Mur83]. Le programme contient m contraintes linéaires d'égalité ou d'inégalité sur $x_1, ..., x_k$, et contient aussi des contraintes dite non négatives $x_1 \geq 0, ..., x_k \geq 0$. La fonction à minimiser (ou à maximiser) est appelée fonction objectif.

Lorsqu'il s'agit de minimiser[1] la fonction objectif étant donnés les coefficients positifs de z, on peut facilement représenter le problème de programmation linéaire par une hiérarchie de contraintes. Les k contraintes non négatives et les m contraintes linéaires d'égalité ou d'inégalité seront représentées dans une hiérarchie comme des contraintes requises. La fonction objectif sera représenté par $z(x_1, ..., x_k) = 0$ et sera considérée comme une contrainte de préférence dans la hiérarchie (puisqu'on connaît *a priori* la borne inférieure (qui est 0) de la valeur de la fonction objectif). Cependant, si la borne inférieure n'est pas connue, cette transformation ne peut pas être fiable. On peut mettre une borne ou un objectif g à la fonction objectif et décider que l'on peut être satisfait si l'on atteint ou si l'on dépasse cet objectif g. Dans ce cas, la fonction objectif est représentée par la contrainte de préférence suivante : $z(x_1, ..., x_k) \leq g$.

Une autre approche serait de représenter la fonction objectif comme la contrainte de préférence $z'(x_1, ..., x_k) = 0$ avec :

$$z'(x_1, ..., x_k) = \begin{cases} -1/z(x_1, ..., x_k) \text{ si } z(x_1, ..., x_k) < -1 \\ z(x_1, ..., x_k) + 2 \text{ si } z(x_1, ..., x_k) >= -1 \end{cases}$$

L'inconvénient de cette approche est de convertir un problème linéaire en un problème non linéaire plus difficile à résoudre. Pour remédier à cet inconvénient, la théorie des hiérarchies de contraintes est étendue. Cette extension consiste à inclure explicitement les fonctions objectifs.

1 Les mêmes arguments décrits pour la minimisation de la fonction objectif peuvent être appliqués lorsqu'il s'agit de maximiser la fonction objectif.

Les fonctions doivent être étiquetées par des étiquettes différentes de l'étiquette *requise*. Les fonctions objectifs $z(x_1, ..., x_k)$ à maximiser sont remplacées par des fonctions objectifs $0\text{-}z(x_1, ..., x_k)$ à minimiser. Soit Z_i l'ensemble des fonctions objectives au niveau i de la hiérarchie. La définition du comparateur *localement-meilleur* est donc étendu comme suit (l'expression $z\theta$ dénote la valeur de $z(x_1\theta, ...,x_k\theta)$ c.à.d la valeur de z lors de l'application de θ à $(x_1, ..., x_k)$).

Définition 3.7 :

Localement-Meilleur$(\theta, \eta, H) \equiv \exists k \in 1...n /$

$\forall i \in 1...k\text{-}1\ g(E(H_i\,\theta)) <>_g g(E(H_i\,\eta)) \wedge (\forall z \in Z_i\ z\theta = z\eta\ ) \wedge$

$(\forall r\ v_r \leq u_r$ avec $v_r \in g(E(H_k\,\theta))$ et $u_r \in g(E(H_k\,\eta))) \wedge$

$(\forall z \in Z_k\ z\theta \leq z\eta\ ) \wedge$

$(\ (\exists j\ v_j < u_j$ avec $v_j \in g(E(H_k\,\theta))$ et $u_j \in g(E(H_k\,\eta))\ ) \vee (\exists z \in Z_k\ z\theta < z\eta\ ))$

avec $g(V) = V$ et $<>_g$ est définit par : $V <>_g U \equiv \forall j\ v_j = u_j$.

En d'autres termes, pour que θ soit *Localement-Meilleur* que η, les effets de θ doivent être identiques à ceux de η pour toutes les contraintes et les fonctions objectives jusqu'au niveau k-1 de la hiérarchie, et au niveau k, θ doit faire mieux que η pour au moins une contrainte ou une fonction objective et mieux ou identique pour tout le reste (i.e. les contraintes et les fonctions objectives du niveau k). Cette extension est dans le même esprit que la définition initiale du comparateur *Localement-Meilleur*, c.à.d. les fonctions objectives et les contraintes sont considérées individuellement.

Le comparateur *Globalement-Meilleur* combine les erreurs d'un niveau donné de la hiérarchie. La borne inférieure des erreurs des contraintes est égale à 0 tandis qu'en général la fonction objectif n'a pas de valeur minimale définie et donc la combinaison de ces deux valeurs en une seule valeur composée ne semble pas très correcte. Par conséquent, lors de l'utilisation des comparateurs globaux, on limite la hiérarchie de contraintes contenant aussi des fonctions objectifs en une hiérarchie telle qu'à chacun de ses niveaux on ait soit des contraintes soit une fonction objectif (dans le cas où il y a plusieurs fonctions objectifs à un niveau, ces fonctions doivent être remplacées par une seule fonction qui combine les différentes valeurs de ces fonctions objectifs). L'extension du schéma du comparateur *Globalement-Meilleur* est défini par :

Définition 3.8 :

Globalement-Meilleur$(\theta, \eta, H, g) \equiv \exists k \in 1...n /$

$\forall i \in 1...k\text{-}1\ g(E(H_i\,\theta)) <>_g g(E(H_i\,\eta)) \wedge (z_i\theta = z_i\eta\ ) \wedge$

$(\ g(E(H_k\,\theta) <_g g(E(H_k\,\eta)) \vee (z_k\theta < z_k\eta\ ))$.

Les relations $<>_g$ et $<_g$ sont équivalentes respectivement à = et < pour les réels, et $g(V)$ est la fonction qui agrège la séquence d'erreurs dans le vecteur V.

Par convention ici, si i est un niveau de la hiérarchie contenant des contraintes, $g(E(H_i\,\theta))$ est défini comme avant et $z_i\theta$ est égal à 0. Tandis que, si i est un niveau contenant une fonction objectif alors $g(E(H_i\,\theta))$ est défini de manière à ce qu'elle soit égale à 0, et $z_i\theta$ est la valeur de la fonction objectif à ce niveau.

3.4 Variation du critère selon les niveaux d'une hiérarchie

Une extension de la théorie des hiérarchies de contraintes a été proposeé par Hiroshi Hosobe dans [HKS+94]. Cette extension consiste à formaliser le cas où les modes de combinaison d'erreur peuvent varier selon les niveaux de la hiérarchie. Par exemple, dans une hiérarchie contenant trois niveaux d'importance, on peut utiliser pour les deux premiers niveaux un critère local (exemple le comparateur *Localement-Prédicat-Meilleur*[1]) pour lequel une solution sera meilleure qu'une autre si elle satisfait un sur-ensemble des contraintes satisfaites par la seconde. Pour le troisième niveau, un comparateur plus fin en général global (par exemple, le comparateur *Moindre-Carrés* ou encore le comparateur *Pire-Cas)* qui impose un ordre total sur les valuations, peut être utilisé pour départager les solutions. Dans ce formalisme, on associe à chaque niveau de la hiérarchie un critère. La solution produite par ce critère est dite *solution type.*

Dans ce mémoire, on se réfère à une *solution type* associée avec le comparateur *Localement-Prédicat-Meilleur* (resp. *Localement-Métrique-Meilleur, Somme-Pondérée-Métrique, Pire-Cas-Metrique, Moindre-Carrés-Métrique*, etc ...) par τ_{LPM} (resp. τ_{LMM}, τ_{SPM}, τ_{PCM}, τ_{MCM}, etc ...) et des contraintes de τ_{LPM} (resp. τ_{LMM}, τ_{SPM}, τ_{PCM}, τ_{MCM}, etc ...) comme des contraintes de *Localement-Prédicat-Meilleur* (resp. *Localement-Métrique-Meilleur, Somme-Pondérée-Métrique, Pire-Cas-Metrique, Moindre-Carrés-Métrique*, etc ...). Dans le chapitre 8 de ce mémoire, nous reviendrons plus en détail sur cette extension.

3.5 Intégration des hiérarchies de contraintes en *PLC*

Récemment, le paradigme des hiérarchies de contraintes a été intégré avec le schéma de programmation logique par contraintes (*PLC*) afin de produire la programmation logique par hiérarchie de contraintes (*PLHC*) [Wil92, DVS+88, BMM+89].La *PLC* et la *PLHC* sont paramétrées par *D* qui est le domaine des contraintes. De plus la *PLHC* est paramétré par le comparateur *C*. Cette intégration de hiérarchies de contraintes à la *PLC* permet à la fondation théorique de la programmation logique d'être complété par l'expressivité des contraintes de préférence. Nous reviendrons en détail sur cet aspect dans la deuxième partie de ce mémoire.

1. Le comparateur *Localement-Prédicat-Meilleur* est une variante de *Localement-Meilleur* utilisant la fonction d'erreur prédicat (qui retourne 0 si la contrainte est satisfaite et 1 sinon)

Synthèse du chapitre

Les différentes parties de ce chapitre présentent les extensions de la théorie des hiérarchies de contraintes. La plupart de ces extensions se manifestent par l'extension des définitions des comparateurs locaux ou globaux.

Les intérêts de ces extensions sont nombreux, par exemple :

la limitation du choix des variables à recalculer pour resatisfaire les contraintes après une perturbation introduite dans le système en intégrant les notations *lecture-seulement* et *écriture-seulement* sur ces variables,

la faculté de pouvoir intégrer des fonctions objectifs dans des hiérarchies de contraintes en redéfinissant les comparateurs,

un remède aux sur-spécifications de certains problèmes (c.à.d ordre total sur les niveaux des hiérarchies) par la définition de l'ordre partiel sur les niveaux des hiérarchies,

la possibilité de faire varier le critère de comparaison selon les niveaux d'une hiérarchie en formalisant le cas où les modes de combinaison d'erreur peuvent varier selon les niveaux de la hiérarchie,

et enfin l'intégration des hiérarchies de contraintes dans le paradigme de la Programmation Logique par Contrainte pour obtenir la Programmation Logique par Hiérarchie de Contraintes.

4 Algorithmes de propagation locale

En préambule de ce chapitre, nous dirons que la recherche d'un algorithme efficace qui pourrait satisfaire tout type de contraintes en utilisant n'importe quel domaine et n'importe quel comparateur serait un essai futile. Ce chapitre donne un aperçu des différents algorithmes existants pour le traitement des hiérarchies de contraintes. Ce chapitre décrit la réponse à la question essentielle posée avant de concevoir tel ou tel algorithme, et qui est : "que fait le système lorsque l'ensemble des contraintes est un ensemble sur-contraint (pas de solution qui satisfasse cet ensemble) ou lorsque l'ensemble des contraintes est un ensemble sous-contraint (il y a plusieurs solutions qui satisfont cet ensemble)". Si le résolveur gère les contraintes d'une application d'interface utilisateur, alors il n'est pas acceptable de signaler une erreur de manipulation. La théorie des hiérarchies de contraintes décrite dans les chapitres précédents indique une voie pour spécifier déclarativement comment le résolveur doit réagir face à cette situation.

Une hiérarchie de contraintes est un ensemble de contraintes où chaque contrainte possède une étiquette qui exprime l'importance de cette contrainte dans le système (ou la préférence de cette contrainte par rapport aux autres contraintes). Etant donnée une hiérarchie de contraintes sur-contrainte, le résolveur peut laisser certaines contraintes parmi les moins préférées non satisfaites pour satisfaire celles qui sont les plus préférées. Si la hiérarchie est sous-contrainte, le résolveur peut choisir n'importe quelle solution. L'utilisateur peut contrôler quelle solution choisir en ajoutant des contraintes étiquetées par des étiquettes faibles. Ces contraintes pourront porter sur les variables pour exprimer le maintien de leurs valeurs (i.e. l'utilisateur exprime via ces contraintes son désir de ne pas changer les valeurs de certaines variables). Ce chapitre discutera aussi un peu plus en détail des algorithmes basés sur la propagation locale. Nous nous attarderons sur quelques aspects intéressants d'ingénierie de certains algorithmes, sur les critères de comparaison utilisés et enfin sur la complexité de ces algorithmes.

Ce chapitre est composé de quatre parties. La première décrit le principe de la propagation locale. La deuxième partie décrit quelques algorithmes existants basés sur la propagation locale pour la résolution de hiérarchie de contraintes fonctionnelles. La troisième partie donne un aperçu sur des algorithmes existants pour la résolution de hiérarchies ayant des contraintes d'égalité et d'inégalité. Et enfin la quatrième partie donne un aperçu sur d'autres algorithmes existants.

4.1 Propagation locale

Parmi les techniques ordinaires pour satisfaire les contraintes, il y a la technique nommée propagation locale. Dans cette technique, la contrainte peut être utilisée pour déterminer la valeur d'une de ses variables au cas où les valeurs de ses $n - 1$ autres variables sont connues. Dans ces conditions, on parle de contrainte à une seule variable de sortie (uni-sortie). La contrainte peut être utilisée pour déterminer les valeurs de m de ses variables si les valeurs de ses $n - m$ autres variables sont connues et si l'on dispose d'une méthode qui calcule ces m valeurs à partir des $n - m$ variables. Dans ces conditions, on parle de contrainte à plusieurs variables de sortie (multi-sorties). Il est bien évident que ces $n - m$ variables peuvent aussi être calculées par d'autres contraintes et ainsi de suite.

La propagation locale est similaire à la propagation des valeurs dans le réseau de contraintes. Cependant, un problème se pose qui est l'existence de contraintes multi-directionnelles dans le réseau et donc il existe plusieurs chemins potentiels de propagation. Le rôle des résolveurs de contraintes en général est de choisir quel chemin prendre. Dans le cas de hiérarchie de contraintes, le chemin à choisir est celui qui doit fournir "*la meilleure*[1] " solution (ou une des meilleures).

Chaque contrainte du réseau possède une ou plusieurs méthodes : il s'agit de petites procédures dont l'exécution d'une seule produit la satisfaction de la contrainte. Chaque méthode détermine la valeur d'une ou plusieurs variables de sortie à partir de celles d'entrée. Par exemple, la contrainte d'addition suivante : $A + B = C$ a trois méthodes possibles : $A \leftarrow C - B$, $B \leftarrow C - A$ et $C \leftarrow A + B$.

L'algorithme de propagation locale produit un chemin de propagation en sélectionnant une méthode de chaque contrainte de la hiérarchie (si la contrainte ne peut pas être satisfaite, cela voudrait dire que l'algorithme n'a pas pu sélectionner de méthode pour cette contrainte). Par conséquent, un chemin produit par l'algorithme de propagation locale utilise au plus une méthode pour déterminer la valeur d'une variable de sortie d'une méthode d'une contrainte. Dans ce cas on considère que toute contrainte possédant une méthode dans ce chemin est satisfaite.

Les résolveurs basés sur la propagation locale (et qui donc ne manipulent que des hiérarchies n'ayant que des contraintes fonctionnelles) sont contraints d'utiliser la fonction d'erreur prédicat qui retourne *0* lorsque la contrainte est satisfaite et *1* sinon (contrairement à la fonction d'erreur métrique). Cette restriction vient du fait que ces résolveurs utilisent les méthodes des contraintes pour les satisfaire.

La propagation locale est incapable de résoudre un ensemble de méthodes de contraintes qui forment un circuit, puisque le chemin produit par ces méthodes sera cyclique, et dans la phase d'exécution de ces méthode on risque de les exécuter un nombre infini de fois.

4.2 Algorithmes pour la résolution d'une hiérarchie de contraintes

Dans les applications interactives, les contraintes d'une hiérarchie viennent souvent s'ajouter graduellement, ceci constitue une raison pour les résoudre par un algorithme incrémental. Un algorithme incrémental maintient une solution courante résolvant les contraintes. Puisque les contraintes sont ajoutées ou retranchées au système, l'algorithme modifie cette solution courante dans le but de trouver une solution qui satisfasse la nouvelle hiérarchie de contraintes.

1. Le mot meilleur ici est relatif au comparateur intégré par le résolveur.

La plupart des algorithmes (exemple : *Blue, DeltaBlue, SkyBlue, QuickPlan* ...) traitant les hiérarchies de contraintes sont basés sur la propagation locale et sur le critère de comparaison *Localement-Prédicat-Meilleur* pour un domaine de contraintes arbitraire où le graphe de méthodes de contraintes ne contient pas de circuit.

Une version incrémentale de l'algorithme *Blue* est connue par le nom *DeltaBlue*. *Blue* et *DeltaBlue* manipulent des hiérarchies de contraintes multi-directionnelles. Les méthodes des contraintes manipulées par ces deux algorithmes ne forment pas de circuits entres elles. Pour l'algorithme *Blue*, les contraintes ne possèdent que des méthodes ayant une seule variable de sortie. Dans le but de surmonter ces restrictions, un algorithme incrémental nommé *SkyBlue* a été conçu. Cet algorithme manipule des contraintes ayant des méthodes possédant plusieurs variables de sortie. Cet algorithme fait aussi appel à un sous-résolveur qui résout les cycles de contraintes lorsqu'ils sont détectés. Dans la suite, on présentera l'architecture de ces algorithmes ainsi que leurs fonctionnalités. On présentera également l'algorithme *QuickPlan* qui manipule les mêmes types de hiérarchies que *SkyBlue*. *QuickPlan* permet de trouver une solution acyclique si elle existe. Ceci nous permettra de situer nos travaux décrits dans les autres chapitres de ce mémoire.

Pour plus de détail sur les algorithmes incrémentaux acycliques, le lecteur est invité à voir [FM89, FMB89, FMB90, Mal91]. [FMB90] et [Mal91] incluent les résultats de complexité et la preuve de complétude. Pour plus de détail sur d'autres algorithmes utilisant la propagation locale, le lecteur est invité à consulter [FW90, FWB92, Wib92, Tro95].

4.2.1 L'algorithme *DeltaBlue*

DeltaBlue est une version incrémentale de l'algorithme *Blue*. La complexité en temps de cet algorithme est de $O(N)$, N étant le nombre de contraintes du système [FW90], tandis que celle de *Blue* est de $O(N^2)$ sur le nombre de contraintes du système. *DeltaBlue* utilise trois catégories de données: les contraintes de la hiérarchie H, les variables contraintes V et la solution courante P. P est aussi appelé "plan", il s'agit d'un graphe orienté acyclique composé de contraintes et de variables. Chaque variable connaît la contrainte qui détermine sa valeur et chaque contrainte sait si elle est actuellement satisfaite et connaît la méthode utilisée (la contrainte utilise une méthode pour sa satisfaction).

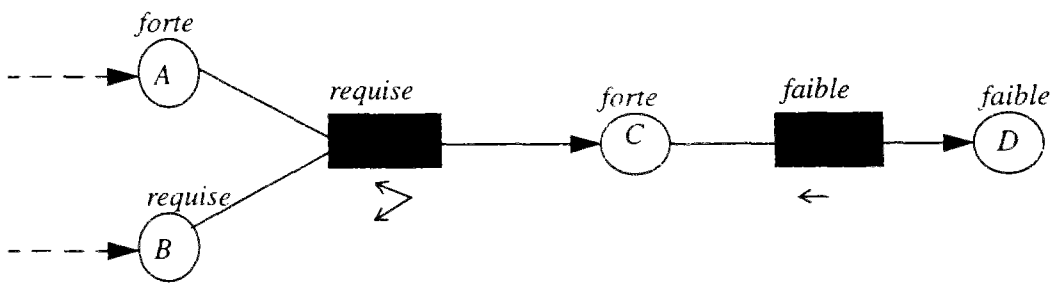
DeltaBlue comporte un programme client qui joue le rôle d'une interface à quatre points d'entrée : *ajout-contrainte*, *ajout-variable*, *enlever-contrainte* et *enlever-variable*. Les variables doivent être ajoutées au graphe avant que la contrainte qui les utilise ne soit introduite. Le retrait d'une variable entraîne le retrait de toute contrainte attachée à cette variable. D'une façon incrémentale, la solution courante P est mise à jour après exécution d'une des deux opérations *ajout-contrainte* et *enlever-contrainte*. Cette algorithme ne résout pas les hiérarchies sous-contraintes. Pour ce faire, une contrainte invisible étiquetée par *très-faible* est attachée à chaque variable quand elle est ajoutée au système. Cette contrainte invisible contraint la valeur de la variable à rester inchangée. Ceci n'a aucun effet secondaire sur la hiérarchie, car cette contrainte invisible est étiquetée par la plus faible étiquette qui puisse exister et donc n'importe quelle autre contrainte fournie par le programme client sera étiquetée par une étiquette plus forte. Cette contrainte invisible sera utilisée seulement si sa variable n'est pas contrainte par la suite (c.à.d. que la variable est sous-contrainte).

L'idée clé de cet algorithme consiste à associer une information suffisante à chaque variable le long de l'exécution de l'algorithme afin de prédire les effets d'un ajout d'une contrainte donnée. Cette prédiction se fera seulement par examen des opérandes (étiquette, variables) de la contrainte ajoutée. Cette information est appelée *étiquette-voyageuse*[1] de la variable et définie par :

La variable v est déterminée par la méthode m de la contrainte c (dans ce cas la variable v est une variable de sortie de la contrainte c). L'étiquette *étiquette-voyageuse* de v est le minimum entre l'étiquette de la contrainte c et les étiquettes associées aux variables d'entrée de la méthode m.

L'étiquette *étiquette-voyageuse* d'une variable peut être vue comme l'étiquette de la plus faible contrainte en amont, c.a.d. la contrainte précédente étiquetée faiblement et qui peut être atteinte via la séquence réversible de contraintes. Ainsi l'étiquette *étiquette-voyageuse* représente l'étiquette de la contrainte faible qui peut être enlevée du système (devient non satisfaite) pour permettre à une autre contrainte d'être satisfaite. Par exemple, la figure 7 montre un graphe de contraintes comprenant quatre variables et deux contraintes. Les variables sont représentées par des cercles et les contraintes par des rectangles. Les variables de sortie d'une méthode sont indiquées par des flèches. Une flèche en pointillé sur une variable indique que la méthode n'est pas sélectionnée pour déterminer cette variable. Un petit diagramme au-dessous de chaque contrainte indique les méthodes non sélectionnées dans la contrainte (méthodes inactives). L'étiquette *étiquette-voyageuse* de la variable D est égale à *faible*. La méthode qui détermine cette variable doit être remplacée par une autre méthode si nécessaire (c.à.d. si par la suite une contrainte portant sur la variable D d'étiquette plus forte que *faible* est ajoutée au système). L'étiquette *étiquette-voyageuse* de la variable C est *forte*, ceci vient du fait que l'*étiquette-voyageuse* de la variable A est *forte* et la contrainte portant sur les variables A et C est étiquetée par *requise* (*requise* est considérée comme supérieure à *forte*).

FIGURE 7 : étiquette-voyageuse



Ajout d'une contrainte par *DeltaBlue*

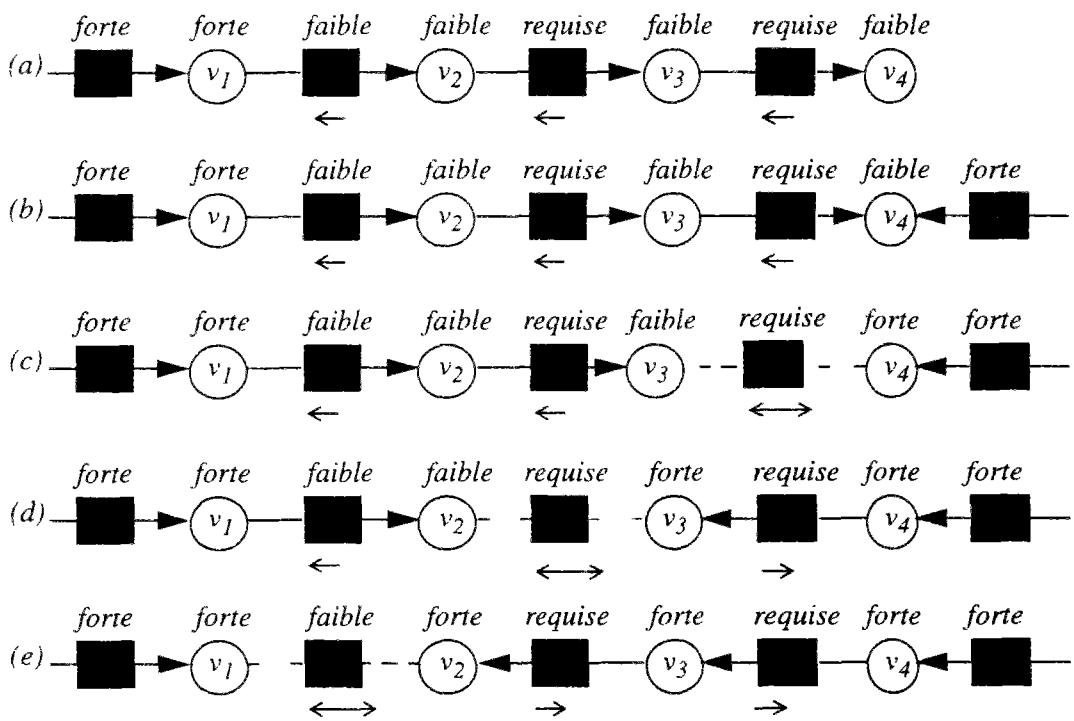
Pour satisfaire une contrainte c ajoutée au système, *DeltaBlue* doit trouver une des méthodes de cette contrainte telle que chacune de ses variables de sortie dans le réseau ait son *étiquette-voyageuse* inférieure à l'étiquette de la contrainte c. Si une telle méthode n'est pas trouvée, alors la contrainte c ne peut pas être satisfaite sans le retrait d'une autre contrainte étiquetée par une étiquette de même importance ou d'importance supérieure à celle de c. L'enlèvement d'une contrainte possédant la même étiquette (même importance) que celle de la contrainte c produira une solution différente que la solution courante mais non meilleure au sens de *Localement-Prédicat-Meilleur*. Dans ce cas, la contrainte c est laissée non satisfaite. L'enlèvement d'une contrainte possédant une étiquette plus forte que celle de la contrainte c ne peut créer qu'une solution plus mauvaise que la solution courante (mauvaise dans le sens où la solution produite satisfait la contrainte c qui est moins importante que la contrainte enlevée du système, ce qui crée une violation de la sémantique du comparateur utilisé). Dans ce cas, la contrainte c est laissée non satisfaite.

1. En anglais, cette appellation est connue par : walkabout strength. Ce nom vient d'une coutume australienne qui dit : "Pour voir plus clair il faut faire une longue marche".

La figure 8 illustre le processus d'ajout d'une contrainte. La figure 8.a montre la situation initiale avant que la contrainte ne soit ajoutée. On observe que l'*étiquette-voyageuse* de la variable v_4 est à faible, ceci est dû à l'étiquette de la contrainte portant sur les variables v_1 et v_2. Dans la figure 8.b, le programme client ajoute une contrainte étiquetée par *forte* et portant sur la variable v_4. *DeltaBlue* considère que cette contrainte doit être ajoutée puisque son étiquette est supérieure à *faible* et par conséquent elle aura comme effet le retrait de la contrainte qui a eu pour effet de mettre la valeur *faible* à l'étiquette *étiquette-voyageuse* de la variable v_4. *DeltaBlue* satisfait la nouvelle contrainte ajoutée au système et retire la contrainte qui déterminait précédemment la variable v_4 (la contrainte portant sur v_3 et v_4) en créant l'état décrit dans la figure 8.c.

Maintenant, du fait que les *étiquette-voyageuse* de v_3 et *étiquette-voyageuse* de v_4 sont moins fortes que celle de la contrainte qui relie ces deux variables, *DeltaBlue* sait qu'il doit resatisfaire cette contrainte. L'algorithme choisit toujours de modifier la variable étiquetée par l'*étiquette-voyageuse* la moins forte. Dans ce cas, c'est la variable v_3 qui est concernée, ainsi l'algorithme resatisfait la contrainte dans la nouvelle direction comme il est montré par la Figure 8.d. Cette étape a pour effet de retirer la contrainte qui détermine la variable v_3. Cette contrainte est alors considérée candidate pour être satisfaite dans une nouvelle direction. Le processus de propagation continue pour atteindre la contrainte responsable de la transmission du contenu de l'étiquette *étiquette-voyageuse* (=*faible*) de la variable initiale. L'algorithme se termine puisque cette dernière contrainte n'est pas assez importante pour qu'elle soit resatisfaite comme il est montré par la Figure 8.e. Si la contrainte initiale à ajouter était étiquetée par *très-faible* alors elle ne serait pas assez importante pour pouvoir déterminer la valeur de la variable v_4 (du fait que la valeur de la variable v_4 est au moins influencée par une contrainte étiquetée par *faible* qui est en tout cas supérieur à l'étiquette *très-faible*).

FIGURE 8 : Ajout d'une contrainte par DeltaBlue

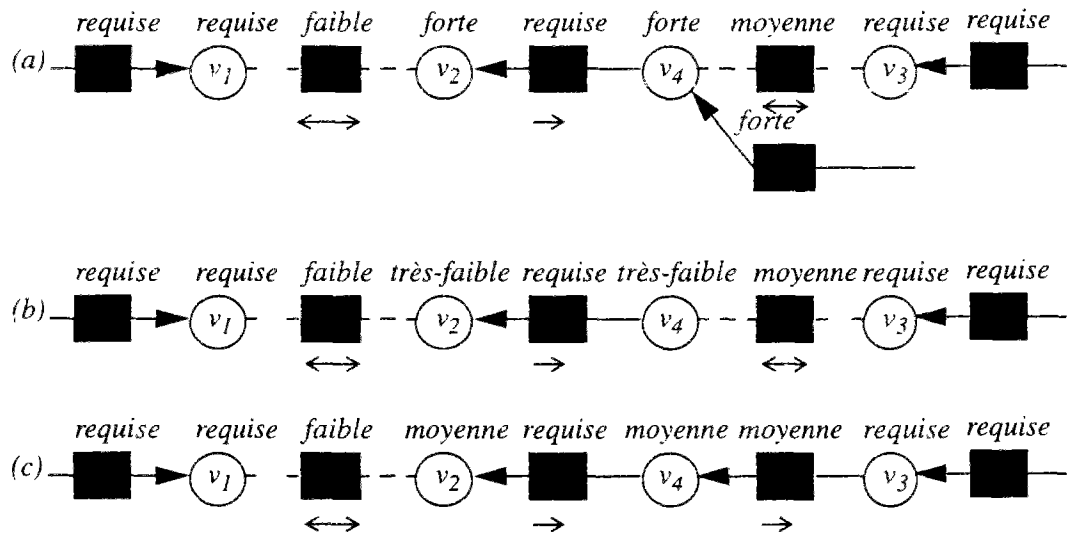


Enlèvement d'une contrainte par *DeltaBlue*

Si la contrainte à enlever est non satisfaite par la solution courante P alors le fait de l'enlever ne modifie en rien la solution courante P. Dans le cas où cette contrainte est satisfaite, son enlèvement du système peut changer le contenu de l'étiquette-voyageuse des variables en aval. Ceci permettra peut être à une ou plusieurs contraintes précédentes non satisfaites de devenir satisfaites. Considérant l'exemple de la figure 9. La figure 9.a montre la situation initiale, le programme client veut retirer la contrainte du côté droit de la variable v_4 étiquetée par *forte*. Il est à noter que la contrainte étiquetée par *faible* et portant sur les variables v_1 et v_2 ainsi que la contrainte étiquetée par *moyenne* portant sur les variables v_3 et v_4 sont toutes les deux non satisfaites puisque leurs étiquettes sont plus faibles que les étiquettes: *étiquette-voyageuse* de v_1, de v_2, de v_3 et celle de v_4. Dans la figure 9.b, la contrainte vient d'être enlevée et l'étiquette *étiquette-voyageuse* en aval vient d'être recalculée puisqu'il n'y a pas de contrainte déterminant la valeur de la variable v_4. L'étiquette *étiquette-voyageuse* de v_4 est maintenant identique à celle de la contrainte invisible introduite avec chaque variable (c.a.d. *très-faible*). Cette étiquette *très-faible* est propagée à travers la contrainte portant sur v_2 et v_3 et affecte l'*étiquette-voyageuse* de la variable v_2.

Maintenant, les deux étiquettes des contraintes non satisfaites sont plus fortes que celles des variables sur lesquelles elles portent et donc ces deux contraintes sont éligibles pour être satisfaites. *DeltaBlue* considère toujours en priorité la contrainte possédant l'étiquette la plus forte. Dans ce cas, la contrainte étiquetée par moyenne est ajoutée et les *étiquettes-voyageuses* de la variable v_2 et celle de la variable v_4 sont recalculées, ceci est illustré dans la figure 9.c. Finalement, la contrainte étiquetée par *faible* portant sur les variables v_1 et v_2 est considérée pour une éventuelle satisfaction. Ceci n'est pas accompli puisque l'étiquette de cette contrainte n'est pas plus forte que les *étiquettes-voyageuses* de v_1 et de v_2. L'algorithme se termine puisqu'il n 'y a plus de contraintes à considérer.

FIGURE 9 : Enlèvement d'une contrainte



Les exemples des figures précédentes contiennent uniquement des contraintes à une seule variable en sortie (c.à.d. que leurs méthodes correspondantes contiennent une seule variable de sortie). Cependant, *DeltaBlue* est capable de manipuler des contraintes ayant plusieurs variables de sortie dans la limite où ces contraintes ne possèdent chacune qu'une seule méthode (on parle de contrainte uni-directionnelle). Il est bien évident que si les contraintes à une seule variable de sortie ne sont pas multi-directionnelles ou si les contraintes à plusieurs variables de sortie ne sont pas uni-directionnelle l'utilisation de l'*étiquette-voyageuse* ne serait pas d'un grand intérêt. Un exemple de contrainte uni-directionnelle, est celui de la contrainte qui capte la position de la souris et détermine la position du curseur sur l'écran. La souris peut altérer la position du curseur tandis que le curseur ne peut pas effectuer un déplacement physique de la souris.

Le principe de *DeltaBlue* est de résoudre seulement les contraintes qui déterminent d'une façon unique les valeurs de leurs variables. Un exemple de contrainte ne déterminant pas d'une façon unique la valeur de sa variable est la contrainte $x > 5$. Il s'agit d'une contrainte de restriction qui ne fixe pas la valeur de sa variable.

Chaque solution générée par *DeltaBlue* est une solution *Localement-Prédicat-Meilleur* à la hiérarchie de contrainte courante. S'il existe un cycle ou un conflit entre des contraintes étiquetées par l'étiquette *requise* (c.à.d. deux méthodes sélectionnées des deux contraintes forment un cycle ou ayant toutes les deux une même variable en sortie) alors l'algorithme s'arrête en générant un message d'erreur.

Pour prouver que les solutions de *DeltaBlue* à des hiérarchies ayant des contraintes qui ne forment pas de cycles sont *Localement-Prédicat-Meilleur*, la notion de contrainte bloquée est définie. Cette notion consiste à dire qu'une contrainte est bloquée si elle est étiquetée par une étiquette considérée supérieure à une des étiquettes de ses variables potentielles de sortie. Ensuite, un lemme est calqué sur cette notion exprimant que s'il n'y a pas de contrainte bloquée dans une solution alors cette solution est *Localement-Prédicat-Meilleur*. La complétude est montrée par récurrence sur la solution courante. Si la solution courante ne possède pas de contrainte bloquée alors l'exécution d'un des quatre points d'entrée du programme ne produit pas de contrainte bloquée. Par exemple, soit une solution P_1 produite par *DeltaBlue*. Supposons qu'il existe une solution P_2 meilleure que P_1 et non produite par *DeltaBlue*. Alors par la définition du comparateur *Localement-Prédicat-Meilleur*, il existe un niveau k de la hiérarchie telle que P_2 satisfait toutes les contraintes que P_1 satisfait et au moins une contrainte c de plus. Soit v_c la variable de c possédant la plus faible *étiquette-voyageuse*, et soit $W_{P1}(v_c)$ l'*étiquette-voyageuse* de la variable v_c dans P_1. Maintenant, puisque c est non satisfaite dans P_1 et P_1 n'a pas de contrainte bloquée, c n'est pas étiquetée par une étiquette plus forte que $W_{P1}(v_c)$. Cependant, puisque c est satisfaite dans P_2, l'étiquette de c doit être plus forte que $W_{P1}(v_c)$. Ce qui constitue une contradiction et donc il ne doit pas y avoir de solution meilleure que P_1.

Contrairement à l'algorithme *Blue* qui réexamine chaque contrainte à la suite d'un ajout ou d'un enlèvement de contrainte au système, *DeltaBlue* utilise la solution courante comme un guide pour trouver la prochaine solution en examinant seulement la contrainte affectée par un changement récent. Dans le cas où le nombre de contraintes affectées par un changement est petit par rapport au nombre de contraintes dans la hiérarchie alors *DeltaBlue* est beaucoup plus rapide que *Blue*. Dans le cas où le nombre de contraintes affectées par ce changement est proche de celui de la hiérarchie alors *DeltaBlue* est moins rapide que *Blue* puisque *DeltaBlue* doit exécuter plus d'opérations pour maintenir sa structure de données.

DeltaBlue est étendu pour obtenir l'algorithme *UltraViolet*. Cette extension consiste à appeler un résolveur de cycle lors de la détection d'un cycle afin de déterminer les valeurs des variables des contraintes ayant les méthodes qui forment ce cycle et ensuite les propager en aval dans le graphe.

4.2.2 L'algorithme *SkyBlue*

Comme nous l'avons vu dans la section précédente décrivant l'algorithme *DeltaBlue*, cet algorithme possède deux restrictions majeures. La première est qu'il est incapable de gérer les contraintes cycliques (si toutefois une hiérarchie contient des contraintes cycliques, *DeltaBlue* s'arrête en générant un message d'erreur). La deuxième restriction est qu'il ne peut pas gérer des contraintes multi-directionnelles ayant plusieurs variables en sortie (c.à.d chaque contrainte peut avoir plusieurs méthodes et chaque méthode peut calculer plusieurs variables).

SkyBlue a été conçu pour remédier à ces deux restrictions. C'est ainsi qu'il permet la construction de cycles et traite les contraintes possédant des méthodes à multi-variables de sortie. *SkyBlue* ne satisfait pas les contraintes qui forment un cycle. Il fait appel à un résolveur plus puissant pour résoudre cette tâche. Cependant, il maintient correctement les contraintes non cycliques du graphe. *SkyBlue* traite les contraintes contenant des méthodes à multi-variables de sortie. Ce genre de contraintes est utilisé dans plusieurs situations. Par exemple, supposons que les variables x et y représentent les coordonnées cartésiennes d'un point et φ et θ représentent les coordonnées polaires du même point. Pour garder ces deux représentations consistantes en parallèle, on aimerait définir une contrainte avec la méthode à 2-variables de sortie x et y : $(x,y) \leftarrow (\varphi \cos \theta, \varphi \sin \theta)$ et une autre méthode à 2-variables de sortie φ et θ : $(\varphi,\theta) \leftarrow (\sqrt{(x^2+y^2)}, Arctang(x,y))$. Les méthodes à multi-variables de sortie sont aussi utilisées pour accéder aux éléments d'une structure de donnée composée. Par exemple, lorsqu'on veut décomposer un objet composé d'un point cartésien à deux variables utilisant une contrainte avec les méthodes : $(x,y) \leftarrow (Point.x, Point.y)$ et $Point \leftarrow Crée\text{-}Point(x,y)$.

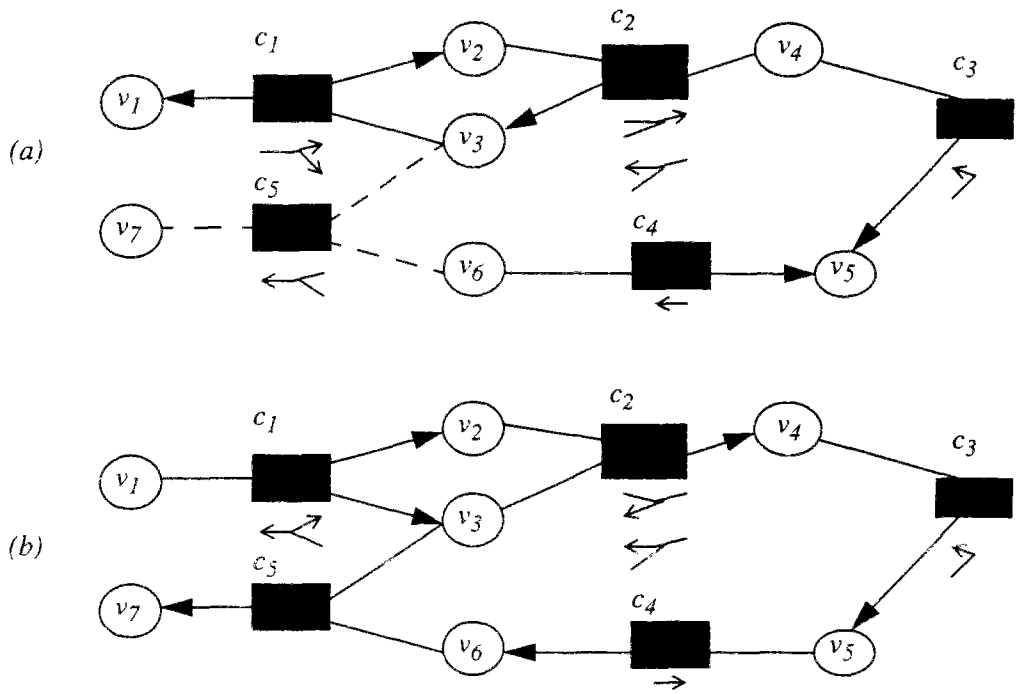
Maloney dans [Mal91] a établi que générer un graphe solution à partir d'un problème de contraintes à sorties multiples était NP-complet. Les auteurs de la ligne *Blue* affirment pourtant que dans la plupart des cas, le complexité en temps n'est pas un problème puisque ces algorithmes changent un sous-graphe de petite taille lors de l'ajout ou du retrait d'une contrainte et donc la complexité en temps est souvent linéaire en nombre de contraintes. Si l'on considère des contraintes uni-variable de sortie non cycliques, *DeltaBlue* est 2 fois plus rapide que *SkyBlue*. Des travaux futurs seront effectués sur *SkyBlue* pour qu'il puisse prédire si la hiérarchie ne contient pas de contraintes cycliques et si elles sont uni-variable de sortie. Dans ce cas, il pourra atteindre les mêmes performances que *DeltaBlue*.

Pour satisfaire un ensemble de contraintes, *SkyBlue* choisit une méthode à exécuter pour chaque contrainte dans cet ensemble. Chaque méthode sélectionnée est considérée comme méthode active dans le graphe. Si un graphe de méthodes contient deux ou plusieurs méthodes actives possédant une même variable de sortie alors il s'agit d'un conflit de méthodes. Dans la figure 10.a, les deux méthodes actives des deux contraintes c_3 et c_4 sont en conflit. *SkyBlue* prohibe le conflit de méthodes en affectant la $n^{ième}$ variable puisqu'il empêche la satisfaction de leurs contraintes correspondantes simultanément. Dans la figure 10.a, si la contrainte c_3 est satisfaite en exécutant sa méthode active (la valeur de la variable v_5 sera déterminée) ensuite la contrainte c_4 est satisfaite en exécutant sa méthode active (la valeur de la variable v_5 sera redéterminée de nouveau). Ceci implique que la contrainte c_3 devienne certainement non satisfaite puisque la valeur de v_5 vient d'être modifiée.

Un graphe de méthodes ne possédant pas de conflit de méthodes ni de cycle peut être utilisé pour la satisfaction des contraintes actives[1] en exécutant ces méthodes dans l'ordre topologique[2]. Par exemple la figure 10.b montre un graphe de méthodes des mêmes contraintes que celles de la figure (a) où toutes les contraintes peuvent être satisfaites en exécutant dans l'ordre les méthodes correspondantes au contraintes c_1, c_2, c_3, c_4 et c_5.

1 On désigne par contrainte active une contrainte possédant une de ces méthodes active dans le graphe.
2 Dans les figures, chaque variable désignée par une flèche est déterminée avant d'être lue.

FIGURE 10 : (a) graphe de méthodes contenant la contrainte inactive c_5, un conflit de méthodes en v_5 et un cycle direct (entre c_1 et c_2). (b) graphe où chaque contrainte est satisfaite.



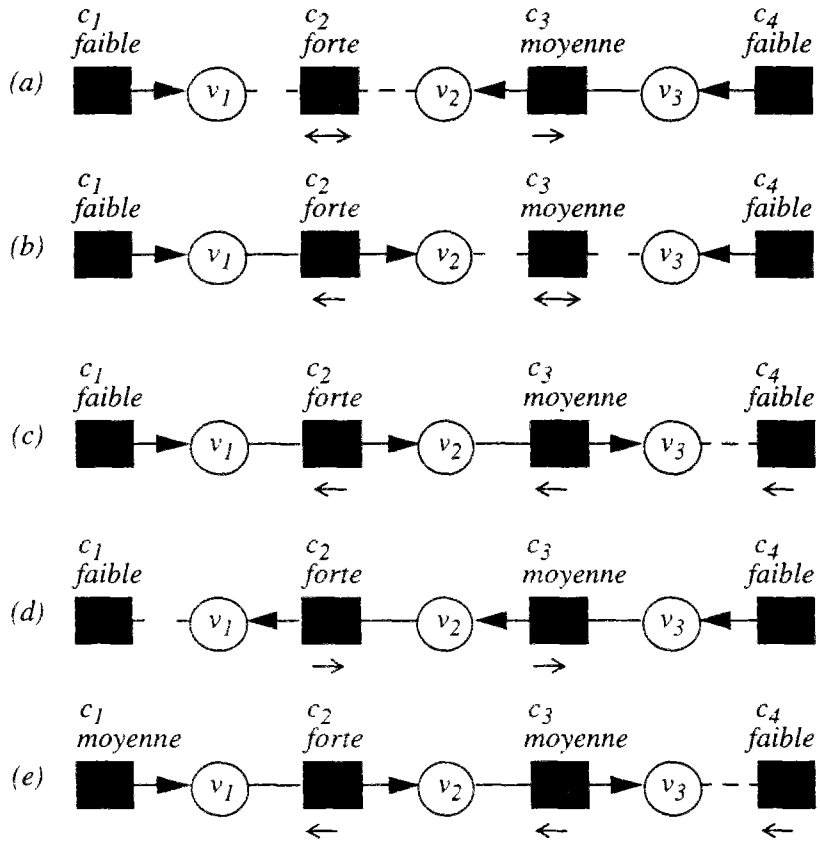
Si un graphe de méthodes contient un cycle direct (comme par exemple entre les deux méthodes des deux contraintes c_1 et c_2 de la figure 10.a) alors il devient impossible de trouver un ordre de tri topologique pour ses méthodes actives. Dans ce cas, *SkyBlue* trie et exécute seulement les méthodes actives en amont du cycle. Toutes les méthodes qui forment un cycle (ou l'aval du cycle) ne sont pas exécutées et leurs variables sont marquées pour spécifier que leurs valeurs ne satisfont pas nécessairement les contraintes actives. Si le cycle est cassé après, les méthodes de ce cycle sont triées et celles en aval sont exécutées correctement.

Traitement d'une hiérarchie de contraintes par *SkyBlue*

SkyBlue utilise les étiquettes des contraintes pour construire un *Graphe-Localement-Meilleur* (*GLM*) de méthodes. Un graphe de méthodes est un *GLM* s'il ne contient pas de conflit entres ses méthodes et s'il ne contient pas de méthode inactive qui puisse être activée en désactivant une ou plusieurs méthodes d'importances inférieures (étiquetées par des étiquettes moins fortes).

Par exemple, considérons le graphe de méthodes de la figure 11.a. Ce graphe n'est pas un *GLM* puisque la contrainte c_2 (étiquetée par *forte*) peut être activée en activant sa méthode qui a pour variable de sortie v_2 et en désactivant la contrainte c_3 (étiquetée par *moyenne*) ce qui produit la figure 11.b. Ce graphe de méthode n'est toujours pas un *GLM* puisque c_3 peut être activée en désactivant la contrainte c_4 en produisant la figure 11.c. Ce dernier est un *GLM* puisque la seule contrainte désactive (c_4) ne peut pas être active en désactivant une contrainte d'importance inférieure (étiquetée par une étiquette inférieure).

FIGURE 11 : Construction de GLM de méthodes



Il peut exister plusieurs *GLM* de méthodes pour un graphe donné de contraintes. Ceci est compatible avec la définition théorique attribuée à l'ensemble S (qui est l'ensemble des valuations qui sont solutions à la hiérarchie). S peut contenir plusieurs valuations puisque les comparateurs utilisés n'établissent pas un ordre total sur les valuations dans l'ensemble S_0.

La figure 11.d montre un autre *GLM* de méthodes qui n'est ni meilleur ni pire que celui de la figure 11.c. *SkyBlue* se contente de construire un des *GLM* possibles d'une façon arbitraire. Si l'on change les étiquettes des contraintes, ceci imposera le choix d'une alternative par rapport à une autre. Par exemple si l'étiquette de la contrainte c_1 est remplacée par *moyenne* alors le seul *GLM* de méthodes doit être celui de la Figure 11.e.

Comme dans *DeltaBlue*, pour contrôler la construction des graphes de méthodes, l'utilisateur ajoute des contraintes invisibles (aussi appelé contraintes de maintien sur les variables). Chacune de ces contraintes possède une seule méthode à une variable de sortie et aucune variable d'entrée. Ces contraintes invisibles (de maintien) expriment la non modification des valeurs de leurs variables. Un type de contraintes similaire est celui des contraintes d'initialisation. Ces contraintes initialisent leurs variables de sortie à des valeurs constantes. Les contraintes d'initialisation peuvent être utilisées pour injecter des nouvelles valeurs à des variables dans un graphe de contraintes. Dans la figure 11, les contraintes c_1 et c_4 sont des contraintes de maintien ou d'initialisation.

Le concept de *lecture-seulement* défini dans le chapitre 3 étend la théorie décrite au chapitre 2 aux contraintes qui ne possèdent pas de méthodes pour déterminer certaines de leurs variables. Comme dans *Blue*, *DeltaBlue* ou *SkyBlue*, les contraintes peuvent ne pas avoir toutes les méthodes de toutes les directions possibles.

Pour certains graphes de contraintes, les *GLM* de méthodes de ce graphe déterminent les solutions du comparateur *Localement-Prédicat-Meilleur*.

SkyBlue maintient les contraintes en construisant un *GLM* de méthodes et en exécutant ces méthodes pour satisfaire les contraintes actives. Initialement, le graphe de contraintes et son *GLM* de méthodes sont vides. *SkyBlue* est invoqué par l'appel d'une des deux procédures d'ajout et de retrait d'une contrainte. Puisque les contraintes sont ajoutées et retranchées au système, *SkyBlue* met a jour incrémentalement le *GLM* de méthodes et exécute ces méthodes pour resatisfaire les contraintes actives.

L'ajout d'une contrainte par *SkyBlue*

Lorsqu'une nouvelle contrainte est ajoutée au graphe de contraintes, il est possible de modifier le graphe de méthodes en activant cette nouvelle contrainte. L'activation de cette nouvelle contrainte se fait par la sélection d'une de ses méthodes. Cette sélection peut créer un changement de méthodes actives des contraintes libellées par des étiquettes plus fortes ou aussi fortes que celle de la nouvelle contrainte. Cette sélection peut aussi désactiver une ou plusieurs contraintes étiquetées par des étiquettes moins fortes que celle de la nouvelle contrainte. Ce processus est connu comme la construction d'une vigne de méthodes. L'ajout d'une contrainte c par *SKyBlue* est effectué par l'exécution des pas suivants :

1. Ajouter la contrainte c inactive au graphe de contraintes et essayer de l'activer en construisant une vigne. S'il n'est pas possible de construire une telle vigne alors laisser la contrainte c inactive (Dans ce cas le graphe est inchangé et donc il est toujours un *GLM*).

2. D'une manière répétitive, essayer d'activer toutes les contraintes non actives dans le graphe en construisant des vignes de méthodes jusqu'à ce qu'aucune des contraintes restantes ne puisse être activée. Il est à signaler que chaque fois qu'une contrainte inactive devient active, une ou plusieurs contraintes faiblement étiquetées sont enlevées. Ces contraintes enlevées sont remises dans l'ensemble des contraintes inactives pour être réessayées après (lors de l'ajout d'une autre contrainte).

3. Exécuter les méthodes sélectionnées (actives) dans le graphe de méthodes pour satisfaire les contraintes actives comme il est décrit précédemment.

Le deuxième pas doit terminer puisqu'il y a un nombre fini de contraintes. Chaque fois qu'une contrainte est enlevée, elle est ajoutée à l'ensemble des contraintes inactives. Ces contraintes ajoutées peuvent être activables après, lors de l'ajout d'autres contraintes étiquetées plus faiblement que ces dernières. Ce processus n'est pas infini et peut s'arrêter éventuellement avec un ensemble de contraintes non activables.

Lorsque le pas 2 se termine, le graphe de méthodes actives résultant est un *GLM* puisqu'aucune vigne supplémentaire ne peut être construite. Par exemple, supposons que l'on vient d'ajouter la contrainte c_2 au graphe de contrainte de la figure 11.a. Une des voies possibles de construction de la vigne est d'enlever la contrainte c_3 et d'activer la contrainte c_2 en choisissant sa méthode qui a pour variable de sortie v_2 (figure 11.b).

Etant donné ce graphe de méthodes, le deuxième pas va essayer de construire une vigne pour activer la contrainte c_3. Ceci est possible en enlevant la contrainte c_4 (figure 11.c). Arrivé à ce point il n'est plus possible de construire une vigne pour activer la contrainte c_4 et par conséquent le pas 2 se termine. Le graphe de méthodes résultant est un *GLM*. Alternativement, si la première vigne avait été construite en enlevant la contrainte c_1 alors le *GLM* de méthodes de la figure 11.d aurait été produit immédiatement et le deuxième pas n'aurait pas été capable d'activer la contrainte c_1.

Le retrait d'une contrainte par *SkyBlue*

La procédure de retrait d'une contrainte est similaire à celle d'ajout d'une contrainte au système. Lorsqu'une contrainte active est retirée du graphe, l'activation d'une ou plusieurs contraintes inactives devient possible. Cela aboutit au même processus de construction de vignes. Le retrait d'une contrainte c par *SKyBlue* est effectué par l'exécution des pas suivants :

1. Si c est inactive, alors son retrait ne permet en aucun cas l'activation d'autres contraintes inactives et donc le graphe de méthode reste un *GLM*.
2. D'une manière répétitive, essayer d'activer toutes les contraintes non actives dans le graphe en construisant des vignes de méthodes (ajouter l'ensemble des contraintes désactivées à l'ensemble des contraintes inactives) jusqu'à ce qu'aucune des contraintes restantes ne puisse être activée. Ce pas se termine en laissant un *GLM* de méthodes.
3. Exécuter les méthodes sélectionnées dans le graphe de méthodes pour satisfaire les contraintes actives.

La construction d'une vigne de méthodes par *SkyBlue*

SkyBlue tente d'activer les méthodes inactives en changeant les méthodes actives des contraintes dans le graphe. Seules les méthodes de contraintes étiquetées par des étiquettes de même préférences ou de préférences supérieures sont concernées par ce changement. Les méthodes des contraintes étiquetées par des étiquettes de préférences strictement inférieures peuvent être inactivées à la suite de cette activation. La technique utilisée par *SkyBlue* est la construction de *vigne* de méthodes (graphe de méthodes) utilisant la recherche en profondeur d'abord en opérant un retour en arrière.

Une vigne est construite en sélectionnant une des méthodes de la contrainte qu'on essaye d'activer (la contrainte racine). Si cette méthode est en conflit avec une méthode d'une autre contrainte active, alors on sélectionne une autre méthode de cette autre contrainte active. Cette dernière peut être aussi en conflit avec d'autres méthodes, etc. Ce processus continue dans le graphe de méthodes et résulte en une vigne de nouvelles méthodes débutant par une méthode de la contrainte racine. Ce processus de construction de vigne se manifeste dans l'un des sens suivants :

1. Si les variables de sortie de la nouvelle méthode sélectionnée dans la vigne ne sont pas déterminées par une autre méthode d'une autre contrainte, alors la branche courante de la vigne s'arrête sur ce point.
2. Si la nouvelle méthode sélectionnée dans la vigne est en conflit avec une méthode d'une contrainte étiquetée faiblement par rapport à la contrainte racine alors cette contrainte est désactivée. Par conséquent, toutes les méthodes d'une vigne appartiennent à des contraintes étiquetées par des étiquettes supérieures ou égales à celle de l'étiquette de la contrainte racine.

3. Si une méthode alternative choisie est en conflit avec une méthode de la vigne alors cette méthode n'est pas ajoutée à la vigne et l'essai d'autres alternatives est effectué. Si toutes les alternatives sont en conflit avec les méthodes de la vigne alors le processus de construction de cette vigne opère un retour arrière . La méthode sélectionnée précédemment est enlevée de la vigne. Cette vigne est ensuite étendue en choisissant d'autres méthodes de ces contraintes. Si aucune méthode de la contrainte racine ne permet la construction d'une vigne sans conflit, alors cette contrainte n'est pas activée.

La figure 12 présente un exemple qui montre le processus de construction d'une vigne. Une vigne complète est un sous-graphe connecté. Une vigne n'est pas nécessairement un arbre puisque les branches séparées peuvent se rencontrer et former un cycle. Si toute méthode de la vigne possède une seule variable de sortie alors la vigne doit avoir la structure d'une tige commençant par la contrainte racine et finissant par des séries d'autres contraintes avec des changements de méthodes sélectionnées. Si des méthodes de la vigne possèdent plusieurs variables de sortie alors la vigne doit être divisée en plusieurs branches avec une branche pour chaque variable de sortie. Les différentes branches ne peuvent pas être étendues indépendamment puisque les méthodes dans les différentes branches ne peuvent pas déterminer les mêmes variables. La recherche par backtrack doit prendre cela en compte en essayant toute combinaison possible de méthodes sélectionnées pour les contraintes dans différentes branches.

Les heuristiques intégrées à SkyBlue

Les performances de *SkyBlue* chutent lorsque le graphe de contraintes devient très large puisque le graphe peut contenir un nombre très important de contraintes inactives que *SkyBlue* va tenter d'activer en essayant de construire les vignes correspondantes. Dans ces conditions, *SkyBlue* exhibe une complexité en temps de $O(M^N)$ avec N est le nombre de contraintes et M est le nombre maximal de méthodes par contrainte.

Dans cette section, on décrira des heuristiques utilisées par *SkyBlue*. Ces heuristiques améliorent les performances de *SykBlue* sur un graphe de contraintes très grand.

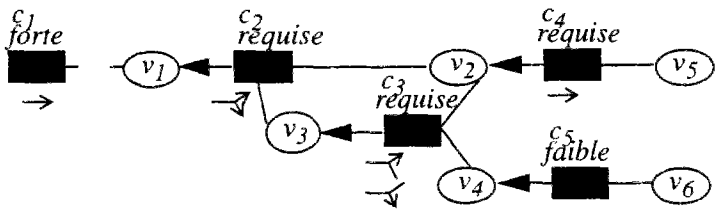
Technique d'assemblage d'étiquettes

Le résultat de chaque appel à l'une des procédures d'ajout ou de retrait de contrainte est un *GLM* de méthodes. Cependant, le graphe de méthode courant doit être un *GLM* si jamais l'une des deux procédures est appelée. Ce fait peut être utilisé pour éviter d'essayer l'activation de certaines contraintes inactives.

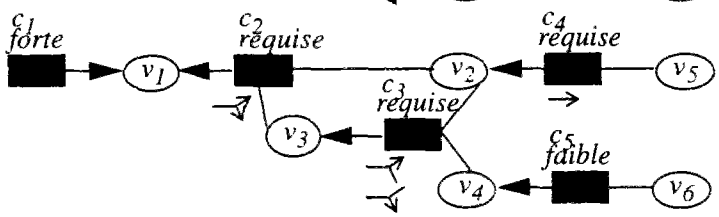
Si à l'issue de l'appel de la procédure d'ajout d'une contrainte c au graphe, cette contrainte est ajoutée effectivement alors il est impossible d'activer une autre contrainte inactive étiquetée par une étiquette de même importance ou d'importance supérieure à celle de c. Puisque, s'il était possible d'activer une telle contrainte après l'ajout de c, il aurait été possible aussi de l'activer avant l'ajout de c et le graphe de méthodes n'aurait pas été un *GLM*.

Après le retrait d'une contrainte c par la procédure de retrait, il est impossible d'activer une contrainte étiquetée par une étiquette plus forte que celle de c. Puisque s'il était possible d'activer une telle contrainte après ce retrait, il aurait été possible de l'activer avant ce retrait et le graphe de méthode n'aurait pas été un *GLM*. Il faut rappeler que la procédure peut permettre à des contraintes étiquetées par la même étiquette ou par des étiquettes moins fortes que celle de c d'être activées.

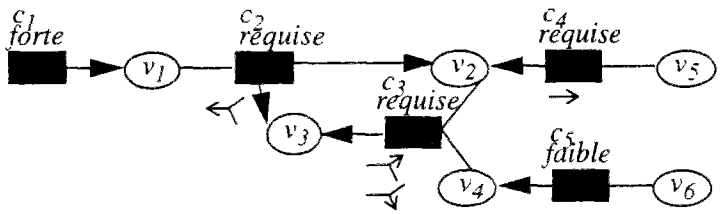
FIGURE 12 : Construction d'une vigne de méthodes



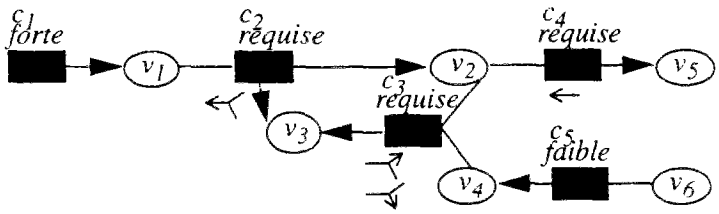
Supposons que l'on commence par ce graphe de méthodes et qu'on veuille activer la contrainte c_1 étiquetée par l'étiquette forte.



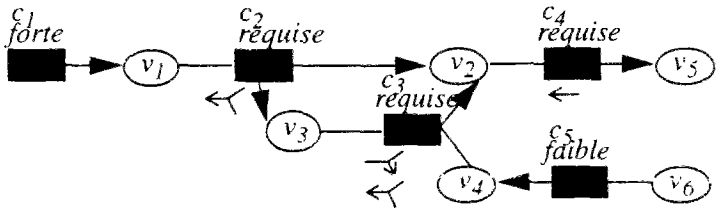
c_1 contient une seule méthode et donc cette méthode est sélectionnée pour être activée. Ceci crée un conflit de méthodes entre c_1 et c_2 et donc une autre méthode de c_2 doit être choisie.



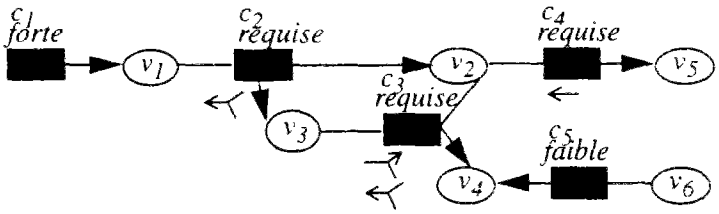
La méthode alternative dans c_2 crée un conflit avec c_3 et un conflit avec c_4.



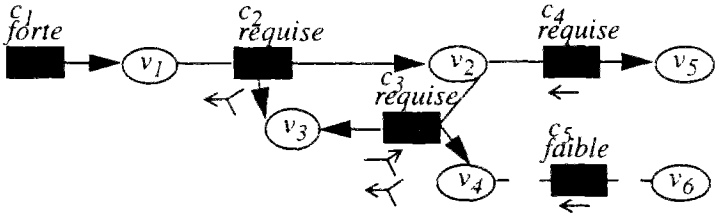
On suppose qu'on procède avec c_4 en premier : on intervertit seulement les méthodes de c_4 et donc la méthode déterminant v_5 est activée. v_5 n'est déterminée par aucune autre méthode et donc le développement de cette branche de la vigne s'arrête en ce point.



On doit considérer une des autres méthodes de c_3. Supposons qu'on essaye celle qui détermine v_2. Cette dernière n'est pas permise puisqu'elle crée un conflit avec c_2 qui se trouve déjà dans la vigne.



Cependant, on opère un retour arrière et on essayer une autre méthode de c_3 Supposons maintenant qu'on essaye la méthode qui détermine la variable v4 (ceci produit un conflit de méthode avec c_5)



Maintenant on considère la contrainte c_5. Puisque cette contraint est étiquetée plus faiblement que la contrainte c_1, on n'a pas à trouver une méthode alternative. On peut tout simplement la supprimer et arrêter la construction de la vigne en ce point.

Technique locale d'assemblage

Si le graphe de méthodes est un *GLM* à la suite d'un ajout ou d'un retrait d'une contrainte *c*, il est clair qu'aucune contrainte inactive non connectée directement à *c* ne sera activée. Il est possible d'être plus sélectif sur les contraintes à activer :

> Lors de l'appel de la procédure d'ajout d'une contrainte *c*, si cet appel termine par la construction d'une vigne qui active la contrainte *c*, il est suffisant de collecter les contraintes inactives qui contraignent les variables en aval dans le graphe. Ces variables sont extraites des variables redéterminées par d'autres contraintes différentes des contraintes initiales.
>
> Après le retrait d'une contrainte *c*, il est suffisant de collecter les contraintes inactives qui contraignent les variables en aval parmi celles déterminées précédemment par *c*.
>
> Si SkyBlue construit avec succès une vigne, les contraintes inactives résultant de cette construction sont ajoutées à l'ensemble des contraintes inactives en examinant les variables en aval redéterminées une nouvelle fois. Comme chacune de ces contraintes est traitée (elle est activée ou il est déterminé qu'elle ne sera pas activée) elle peut être enlevée de l'ensemble. Lorsque cet ensemble devient vide, il n'y a plus de contrainte inactive qui puisse être activée.

Une technique identique peut être utilisée pour réduire l'ensemble des méthodes à exécuter. Au lieu d'exécuter toutes les méthodes sélectionnées des contraintes actives dans le graphe des contraintes, il est seulement nécessaire de collecter et d'exécuter les méthodes sélectionnées des nouvelles contraintes activées et les méthodes en aval des variables redéterminées.

L'étiquette-voyageuse

La vigne est construite par le processus répétitif qui est la sélection d'une nouvelle méthode d'une contrainte et l'essai d'étendre la vigne en partant des variables de sortie de cette nouvelle méthode. Cette extension n'est réalisée que s'il n'y a pas de conflit entre les différentes branches et que la vigne rencontre des variables non déterminées par des méthodes de contraintes étiquetées par des étiquettes identiques ou supérieures à celle de la contrainte racine. Si *SkyBlue* peut prédire qu'une de ces conditions est fausse, alors la méthode sélectionnée doit être rejetée immédiatement sans essayer l'extension de la vigne.

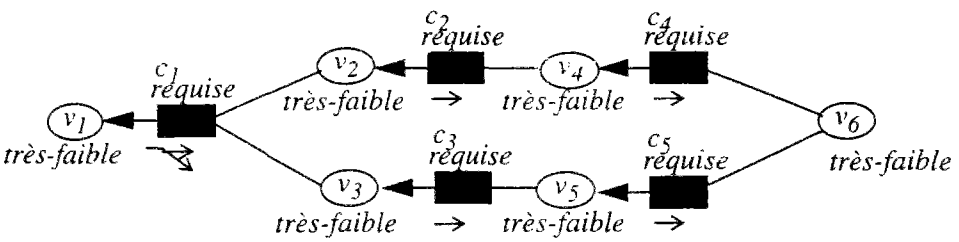
Nous avons vu que le résolveur *DeltaBlue* prédit l'activation d'une nouvelle contrainte introduite au système en utilisant le concept de l'*étiquette-voyageuse*. L'*étiquette-voyageuse* d'une variable indique l'étiquette de la contrainte qui doit être retirée (ne servira pas à déterminer cette variable) pour que cette variable puisse être déterminée par une autre contrainte. L'*étiquette-voyageuse* d'une variable peut être l'étiquette de la contrainte courante qui détermine cette variable, ou bien elle peut être la plus faible étiquette d'une contrainte dans la branche partant de cette variable. Cette contrainte peut être amenée à être retirée lors d'une resélection de méthodes. Si la variable n'est déterminée par aucune des contraintes du système, son *étiquette-voyageuse* est assignée à *très-faible*[1].

1 Cette étiquette est considérée comme la plus faible étiquette de toutes les étiquettes du système.

Une variable peut aussi porter cette étiquette si elle est laissée indéterminée lorsque le résolveur resélectionne les méthodes sans enlever aucune contrainte (une autre interprétation attribuée à l'étiquette *très-faible* est que chaque variable a un état implicite préféré par *très-faible* qui spécifie que la valeur de la variable peut être changée si une contrainte étiquetée par une étiquette plus forte la détermine).

Comme nous l'avons mentionné auparavant, une propriété importante de l'algorithme *DeltaBlue* concernant le calcul de l'*étiquette-voyageuse* est qu'elle peut être déterminée en utilisant les informations locales. L'*étiquette-voyageuse* d'une variable peut être calculée à partir de l'étiquette de la contrainte qui la détermine et des *étiquettes-voyageuses* des autres variables de cette contrainte. Si le graphe de méthodes ne possède pas de cycle[1], toutes les *étiquettes-voyageuses* peuvent être mises à jour en associant les *étiquettes-voyageuses* des variables non déterminées à *très-faible* et en traitant chaque contrainte active dans un ordre topologique (afin d'initialiser les *étiquettes-voyageuses* des variables déterminées).

FIGURE 13 : étiquettes-voyageuses en SkyBlue



Cette technique de calcul n'est plus valable dans *SkyBlue* puisqu'on considère des contraintes possédant des méthodes ayant plusieurs variables de sortie. Considérons le graphe de méthodes de la figure 13. *DeltaBlue* doit correctement calculer les *étiquettes-voyageuses* des variables v_2 ... v_6 (dans ce cas, elles sont égales à *très-faible*).

Mais qu'en est-il de l'*étiquette-voyageuse* de la variable v_1? L'*étiquette-voyageuse* de v_2 et celle de v_3 impliquent que celle de v_1 doit être à *très-faible* puisque la méthode alternative de la contrainte c_1 peut être choisie pour déterminer les variables v_2 et v_3. Cependant, il n'est pas possible pour cette méthode de déterminer ces deux variables simultanément sans révoquer une contrainte requise du graphe. Une simple resélection de méthode résultera en un conflit en v_6 entre la contrainte c_4 et la contrainte c_5. Cette information ne peut pas être connue sans explorer le graphe. Ceci conduit à perdre le gain apporté par cette technique d'*étiquette-voyageuse* (c.à.d. qu'on pouvait les calculer en utilisant uniquement les informations locales).

Dans *SkyBlue*, la définition de l'*étiquette-voyageuse* est modifiée. L'*étiquette-voyageuse* d'une variable est définie par la borne inférieure sur l'étiquette de la contrainte la plus faible qui devrait être enlevée afin de permettre à cette variable d'être déterminée par une nouvelle contrainte. *SkyBlue* utilise cette nouvelle définition pour rejeter des méthodes lors de la construction d'une vigne : si une des variables de sortie d'une méthode a son *étiquette-voyageuse* égale ou supérieure à l'étiquette de la contrainte racine. Dans ce cas, il n'est pas possible de compléter la vigne en utilisant cette méthode. L'utilisation de cette technique ne peut pas éliminer tous les retours-arrière durant la construction d'une vigne mais elle les réduit considérablement.

1. Cette condition est requise pour *DeltaBlue*.

Lors de la construction d'une vigne avec succès, le graphe de méthodes est donc modifié et les *étiquettes-voyageuses* doivent être mises à jour pour correspondre aux nouvelles méthodes dans le graphe. Ceci est réalisé en considérant toutes les méthodes actives dans le graphe dans un ordre topologique et en recalculant ces *étiquettes-voyageuses* des variables déterminées par ces méthodes. Il est possible d'appliquer *la technique locale d'assemblage* décrite auparavant à cette situation en traitant seulement les contraintes actives en aval des variables redéterminées.

SkyBlue utilise cette nouvelle définition de l'*étiquette-voyageuse* pour simplifier le traitement des cycles. Si le graphe de méthodes contient des cycles, il n'est donc pas possible de trouver un ordre de tri topologique pour les contraintes. Les *étiquettes-voyageuses* des variables dans le cycle doivent être calculées en examinant toutes les contraintes dans le cycle ce qui nécessite un calcul non local. Au lieu de procéder de cette façon, *SkyBlue* choisit une méthode dans le cycle et calcule les *étiquettes-voyageuses* de ces variables de sortie comme si toutes les *étiquettes-voyageuses* de ces variables d'entrées étaient à *très-faible*. Ceci garantit une borne inférieure et simplifie la mise à jour des *étiquettes-voyageuses* par rapport au coût croissant de la recherche pendant la construction d'une vigne puisque les *étiquettes-voyageuses* en aval dans le cycle peuvent être étiquetées plus faiblement que nécessaire.

4.2.3 L'algorithme *QuickPlan*

Comme nous l'avons vu, les sections précédentes ont décrit les algorithmes *DeltaBlue et SkyBlue*. Ces algorithmes possèdent deux défauts majeurs. Le premier est qu'ils ne garantissent pas une solution acyclique si elle existe. C'est-à-dire, supposons que la hiérarchie manipulée contient des solution cycliques et au moins une solution acyclique. *DeltaBlue* comme *SkyBlue* ne garantissent pas de la trouver. Il suffit que *DeltaBlue* (resp. *SkyBlue*) parte sur la construction d'une chaîne de méthodes formant un cycle et lors de la construction de ce cycle le message d'erreur "*Contraintes Formant un Cycle*" apparaît et il s'arrête (resp. appelle un sous-résolveur de cycle et il continue). *DeltaBlue* comme *SkyBlue* ne cherchent pas à déterminer s'il y a existence d'une autre solution acyclique. Le deuxième défaut est que dans le pire des cas ces algorithmes acquièrent une complexité en temps exponentielle pour trouver une solution à une hiérarchie de contraintes à plusieurs variables de sorties. L'algorithme *QuickPlan* a été conçu pour remédier à ces deux défauts. *QuickPlan* garantit de trouver une solution acyclique si elle existe en $O(N^2)$ (N est le nombre de contraintes).

Dans la suite de cette section, nous présentons des vues générales sur les algorithmes de *QuickPlan*. Seules les idées clés de ces algorithmes suivies d'exemples seront présentées. Les versions complètes de ces algorithmes ainsi qu'une version incrémentale sont dans [Van95]. On présentera en premier une vue sur l'algorithme manipulant des contraintes à une seule variable de sortie. Ensuite, on présentera l'extension de cet algorithme afin de manipuler des contraintes ayant plusieurs variables en sortie. Et enfin, on présentera la résolution d'une hiérarchie de contraintes par utilisation du comparateur *Localement-Prédicat-Meilleur*.

Propagation de degré de liberté et résolution d'un ensemble de contraintes à une seule variable de sortie.

QuickPlan est basé sur la propagation de degré de liberté [Sut63a, Bor81, Van88]. Cette technique manipule un ensemble de contraintes non satisfaites et consiste à :

1- chercher une variable dans le graphe telle que :

cette variable est une variable de sortie d'une méthode d'une contrainte et en plus,

cette variable doit être attachée à une seule contrainte.

2- pour cette variable,

sélectionner la méthode (pour satisfaire la contrainte) qui calcule cette variable,

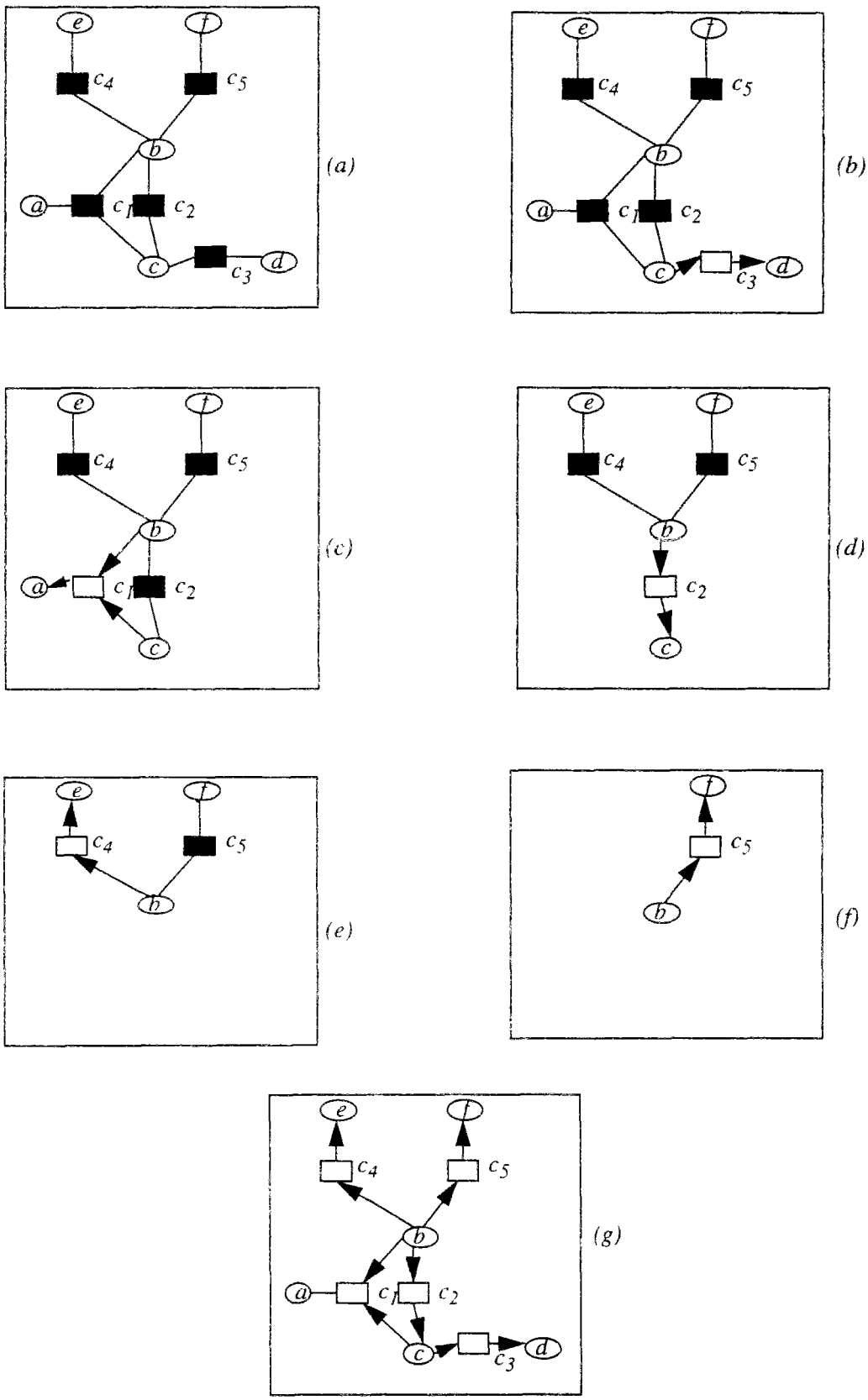
enlever la contrainte possédant cette méthode de l'ensemble des contraintes non satisfaites.

3- retourner au pas 1.

Cet algorithme se termine lorsqu'il ne reste plus de contraintes dans l'ensemble des contraintes non satisfaites ou lorsqu'une variable est attachée à plus d'une contrainte. Dans le cas favorable, le graphe de méthode constitue une solution acyclique. Les contraintes sont satisfaites en exécutant tout simplement cet ensemble de méthodes dans un ordre topologique. Dans le cas défavorable, le sous-graphe restant est considéré cyclique puisqu'il n'y a pas de possibilité de diriger le flot sans aboutir à un cycle entres les méthodes de ce sous-graphe.

Par exemple, considérons la figure 14 qui montre le fonctionnement de cet algorithme. Dans la figure 14.a, la variable *d* est attachée à une seule contrainte (c_3). Dans un premier temps, l'algorithme sélectionne la méthode qui calcule la variable *d* de cette contrainte (figure 14.b). Ensuite, il élimine la contrainte c_3 ainsi que ses arrêtes du graphe (figure 14.c). Ces deux pas sont répétés jusqu'à ce que toutes les contraintes soient éliminées du graphe (figure 14.c-14.f). Le résultat est le graphe acyclique de la figure 14.g.

FIGURE 14 : Résolution d'un graphe par propagation de degrés de liberté.



Résolution d'un ensemble de contraintes ayant plusieurs variables de sortie

L'extension de l'algorithme précédent pour la résolution des contraintes ayant plusieurs variables de sortie est très simple à réaliser. Cette extension est décrite par la procédure ci-dessous :

1- chercher un ensemble de variables dans le graphe tel que :

ce sous ensemble est attaché à une seule contrainte et,

ce sous ensemble constitue les variables de sortie d'une méthode de cette contrainte

2- identifier la méthode sélectionnée qui calcule cet ensemble de variables dans la contrainte,

enlever la contrainte possédant cette méthode de l'ensemble des contraintes non satisfaites.

3- retourner au pas 1.

Le pas 1 peut être réalisé en procédant comme suit :

1'- chercher une variable dans le graphe telle que :

cette variable est une variable de sortie d'une méthode[1] d'une contrainte et en plus,

cette variable doit être attachée à une seule contrainte.

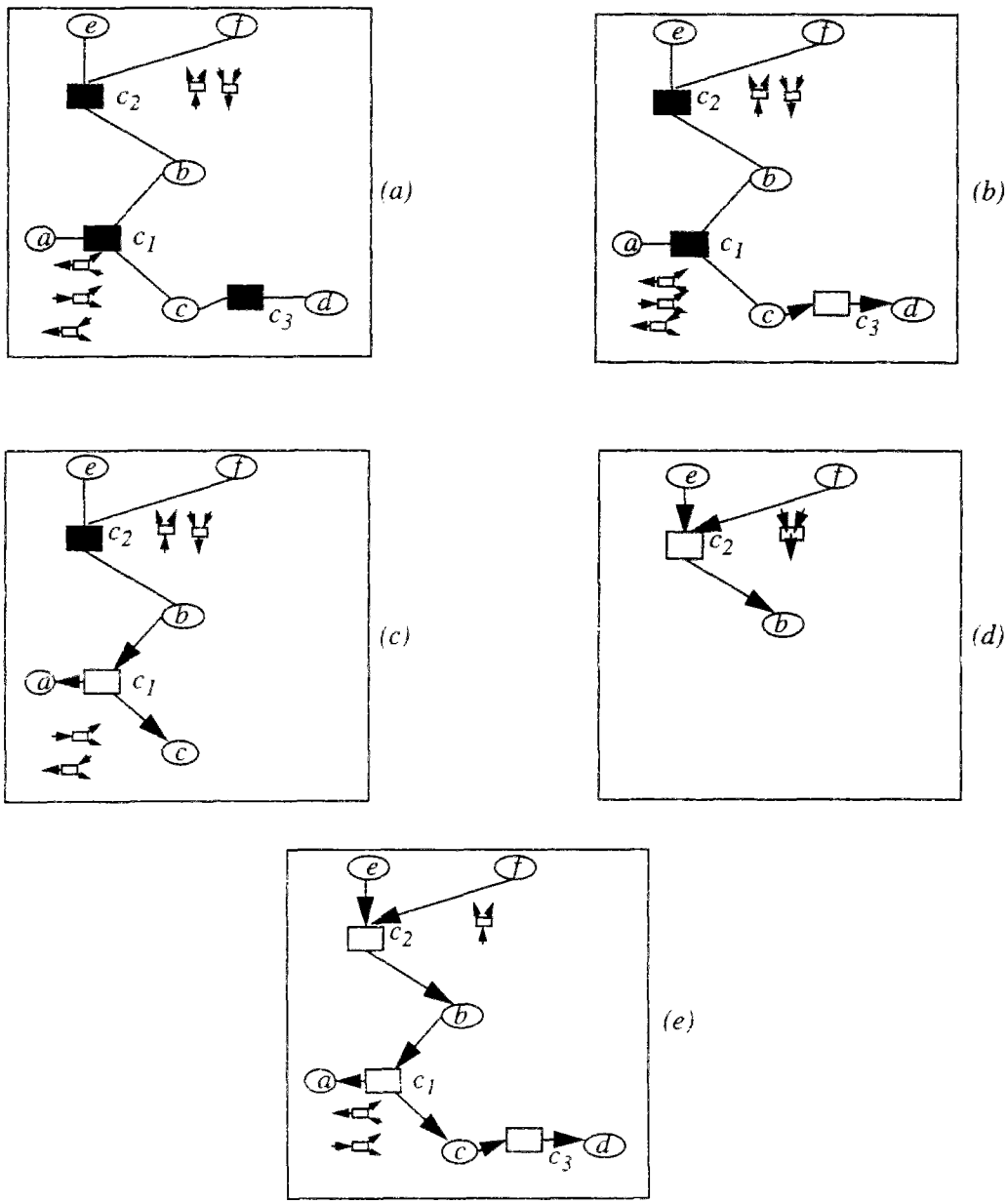
2'- s'il existe une variable de l'ensemble des autres variables de sortie de cette méthode qui n'est pas attachée à une seule contrainte alors retourner au pas 1'.

Ce nouvel algorithme obtenu se termine s'il ne reste aucune contrainte dans le graphe (dans ce cas la solution obtenue est une solution acyclique) ou s'il reste un ensemble de contraintes formant un cycle.

Par exemple, considérons la figure 15 qui montre le déroulement de ce nouvel algorithme obtenu à partir du premier. Dans la figure 15, les méthodes iconifiées représentent les méthodes non sélectionnées pour la satisfaction des contraintes. Dans la figure 15.a, la variable d est attachée à une seule contrainte (c_3). L'algorithme sélectionne la méthode qui calcule la variable d de cette contrainte (figure 15.b). Ensuite, il élimine la contrainte c_3 ainsi que ses arêtes du graphe (figure 15.c). Arrivé à ce stade, l'algorithme sélectionne la variable a. Il existe deux méthodes dans la contrainte c_1 pour lesquelles la variable a est une variable de sortie. La première méthode a pour variables de sortie les variables a et c et la deuxième méthode a pour variables de sortie les variables a et b. L'algorithme choisit la première puisque, pour la deuxième, la variable b est attachée à deux contraintes (condition du pas 2'). La première méthode est identifiée et la contrainte c_1 ainsi que ses arrêtes sont éliminées du graphe (figure 15.d). L'algorithme choisit ensuite la variable b et identifie la méthode qui détermine cette variable (figure 15.d). Après ce pas, il ne reste plus de contrainte dans le graphe et, par conséquent, l'algorithme se termine. Le graphe de méthodes dans la figure 15.e constitue une solution acyclique du graphe de contraintes initial.

1. S'il y a plusieurs choix de méthodes, l'algorithme sélectionne la méthode qui possède le plus petit nombre de variable en sortie. Ceci afin de maximuser le nombre de contraintes satisfaites.

FIGURE 15 : Résolution d'un graphe de contraintes ayant plusieurs variables en sortie



Résolution d'une hiérarchie de contraintes.

Pour la résolution d'une hiérarchie de contraintes où les contraintes sont étiquetées, *QuickPlan* modifie légèrement la technique décrite dans le paragraphe précédent. Si l'algorithme rencontre un sous-graphe dans lequel chaque variable est attachée à plusieurs contraintes, au lieu de terminer, il supprime la contrainte ayant l'étiquette la plus faible pourvu qu'elle ne soit pas requise. L'algorithme continue à satisfaire le nouveau sous-graphe obtenu. L'algorithme alterne les deux pas : suppression et résolution jusqu'à ce qu'il ne reste plus de contrainte dans le graphe ou jusqu'à ce que chaque variable du sous-graphe restant ne soit attachée qu'à des contraintes requises.

La preuve de la terminaison de l'algorithme est construite avec les même arguments que celles présentées auparavant. Si l'algorithme résout successivement les contraintes et en supprime quelques unes alors la solutions générée n'est pas forcément une solution meilleure au sens *Localement-Prédicat-Meilleur.*

Pour obtenir une solution qui est *Localement-Prédicat-Meilleur, QuickPlan* réessaie les contraintes éliminées par ordre décroissant sur leurs étiquettes. C'est ainsi qu'il peut satisfaire certaines de ces contraintes en exécutant l'algorithme de propagation de degré de liberté sur l'ensemble obtenu par l'union des contraintes satisfaites précédemment et la contrainte qu'il réessaie de satisfaire.

L'essai effectué par l'algorithme de propagation de degrés de liberté, qui consiste à satisfaire une contrainte éliminée, peut créer l'élimination d'une ou plusieurs contraintes ayant des étiquettes inférieures à celles de la contrainte essayée. Cependant, l'algorithme ne doit pas éliminer une contrainte étiquetée par une étiquette égale ou supérieure à celle de la contrainte essayée, puisque ceci conduirait à une solution moins bonne ou, au mieux, aussi bonne que la précédente. Par conséquent, lorsque l'algorithme atteint le point où il doit éliminer une contrainte étiquetée par une étiquette plus forte ou égale à celle de la contrainte réessayée pour avancer, il doit terminer et communiquer à *QuickPlan* que la contrainte qu'il a essayée ne peut pas être satisfaite. Tandis que si l'algorithme réussit à satisfaire la contrainte qu'il a essayée, alors toute contrainte éliminée à cause de cette satisfaction est remise dans la liste des contraintes qui doivent être réessayées.

Puisque *QuickPlan* tente de satisfaire les contraintes éliminées selon l'ordre décroissant de leurs étiquettes, alors chaque contrainte éliminée est essayée une seule fois. La hiérarchie contient un nombre fini de contraintes et par conséquent *QuickPlan* se termine. La solution générée est une solution *Localement-Prédicat-Meilleur.*

4.3 Algorithmes de résolution de contraintes d'égalité et d'inégalité

Un désavantage des algorithmes existants utilisant le principe de la propagation locale est l'incapacité de résoudre des contraintes cycliques. Dans certains cas, ces algorithmes peuvent trouver une solution acyclique si elle existe mais ceci n'est pas garanti. Ces algorithmes s'arrêtent souvent en générant le message d'erreur "*Graphe de contraintes cyclique*". De plus, si les contraintes sont simultanément vraies (c.à.d. deux contraintes sont en conflit sur une variable, et il existe une valeur pour cette variable qui satisfait ces deux contraintes), ces algorithmes ne peuvent pas les résoudre.

Un autre ensemble d'algorithmes existe. Ces algorithmes peuvent résoudre les hiérarchies contenant une collection arbitraire de contraintes linéaires d'égalité et d'inégalité. Ces algorithmes sont basés sur les comparateurs suivants : *Localement-Métrique-Meilleur* (τ_{LMM}), *Somme-Pondérée-Métrique*(τ_{SPM}), *Cas-Pire-Métrique*(τ_{CPM}), *Moindre-Carrés-Métrique*(τ_{MCM}). Ces comparateurs utilisent le domaine métrique pour compter les erreurs produites par les valuations sur les contraintes. Ces algorithmes sont des instances de l'algorithme *DeltaStar* [FW90, FWB92] et sont collectivement référencés comme les algorithmes *Orange.*

DeltaStar est un algorithme incrémental qui résout une hiérarchie de contraintes. Il est basé sur une alternative équivalente à la description de la théorie des hiérarchies de contraintes. La preuve de cette équivalence est décrite dans [Wil92, FW90, Fre91]. La principe de la théorie des hiérarchies de contraintes est décrit dans les chapitres précédents et attaché à la dichotomie entre les niveaux des contraintes dure et de préférence, alors que celui de la théorie alternative est attaché au raffinement hiérarchique de l'ensemble de solutions.

Orange est une série de trois algorithmes utilisant le principe de *DeltaStar* en transformant une hiérarchie de contraintes en une série de problèmes de programmation linéaire. L'ensemble de contraintes d'un niveau donné de la hiérarchie est transformé en un programme linéaire qui est résolu par l'algorithme du simplexe. Ces algorithmes sont spécialisés pour la recherche d'une ou plusieurs solutions d'une hiérarchie contenant des contraintes d'égalité ou d'inégalité. Aucun de ces trois algorithme ne traite les hiérarchies à ordre partiel sur les niveaux ni les annotations de *lecture-seulement* ou *écriture-seulement* sur les variables.

4.4 Autres algorithmes

Bien qu'elles ne soient pas destinées à résoudre des hiérarchies de contraintes, plusieurs autres techniques de résolution sont disponibles comportant la relaxation, la recherche de solution dans un domaine fini et la réécriture d'équation [KJ84, Bor81, Lel87]. Cette dernière est empruntée aux langages de programmation fonctionnelle avec ajout de support pour les contraintes multi-directionnelles et pour les objets.

Mackworth [Mac77], Van Hentenryck [Van89] et plusieurs autres auteurs décrivent des algorithmes efficaces pour la résolution d'un ensemble de contraintes où les variables ont un domaine de valeurs fini. Ces algorithmes considèrent la relaxation comme une technique numérique itérative dans laquelle chaque valeur de variable est ajustée pour minimiser l'erreur dans le processus de satisfaction des contraintes qui lui sont attachées.

Red est un algorithme utilisant un système de généralisation de réécriture de graphe basé sur *Bertrand* [Lel86]. Cet algorithme est capable de résoudre des hiérarchies de contraintes cycliques ainsi que des équations simultanées.

Yellow est un algorithme utilisant la relaxation classique qui produit une solution de moindre carré [Sut63b]. La complexité en temps de cet algorithme est supérieure à celle de *Orange*. Cependant, il peut traiter des hiérarchies contenant des équations non-linéaires.

Green est un algorithme de résolution de contraintes portant sur des variables ayant des domaines finis. Cet algorithme combine la propagation locale et la technique de génération et test de l'arbre de recherche.

Synthèse du chapitre

Ce chapitre présente une vue générale sur la plupart des résolveurs existants pour la résolution ou le maintien de la cohérence entre les contraintes d'une hiérarchie. Ces algorithmes permettent d'obtenir une ou plusieurs valuations de l'ensemble S des solutions tel qu'il a été construit d'une façon théorique dans le chapitre 2.

Nous avons essentiellement exposé dans ce chapitre plusieurs idées qui nous ont semblé intéressantes et qui sont utilisées dans les algorithmes (*DeltaBlue*, *SkyBlue* et *QuickPlan*) manipulant des contraintes fonctionnelles. Ces algorithmes sont basés sur la propagation locale et implémentent tous le comparateur *Localement-Prédicat-Meilleur*.

Le tableau de la figure 16 est une synthèse sur les fonctionnalités des quatre algorithmes étudiés de près dans ce chapitre.

FIGURE 16 : Comparaison entre résolveurs basés sur la propagation locale[1].

utilise / résolveurs	contraintes ayant : plusieurs variables en sortie	contraintes sont : multi-directionnelles	contraintes sont dans : une hiérarchie	compatible avec résolveur de cycle	garantit solution acyclique	complexité dans le pires de cas
Blue	–	+	–	–	–	O(N)
DeltaBlue	+ –	– +	+	–	–	O(N)
SkyBlue	+	+	+	+	–	$O(M^N)$
QuickPlan	+	+	+	+	+	$O(N^2)$

1. Concernant *DeltaBlue*, le carré en pointillé dans cette figure indique que si les contraintes possèdent plusieurs variables en sortie alors elles ne doivent pas être multi-directionnelles.

5 Un nouveau résolveur : Houria

Après avoir décrit dans le détail la théorie des hiérarchies de contraintes ainsi que certaines extensions qui lui sont apportées, nous avons examiné au chapitre 4 certaines fonctionnalités d'une série de résolveurs existants pour la résolution de ces hiérarchies de contraintes. Chacun de ces résolveurs est basé sur un critère de comparaison de valuations. La plupart de ces derniers sont des algorithmes de propagation locale manipulant des hiérarchies de contraintes fonctionnelles et utilisant des erreurs binaires et travaillant en deux passes. La première consiste à planifier (en tenant compte d'un critère de comparaison) un graphe orienté de méthodes sans conflits et souvent sans circuits. Et la deuxième passe consiste à exécuter les méthodes de ce graphe planifié. Ainsi une valuation qualifiée de solution respectant le critère utilisé est obtenue.

La plupart des algorithmes de propagation locale manipulant des hiérarchies de contraintes fonctionnelles utilisent un mode de comparaison local assez simple entre les différentes valuations : une valuation sera meilleure qu'une autre s'il existe un niveau pour lequel elle satisfait un sur-ensemble des contraintes de la seconde, et si pour tous les niveaux plus importants, elles satisfont toutes deux les mêmes ensembles de contraintes. D'après cet ordre local, deux valuations qui satisfont deux ensembles de contraintes non inclus l'un dans l'autre au sein d'un même niveau de la hiérarchie sont incomparables. Dans la plupart des cas, cette incapacité à comparer ces valuations est une situation indésirable puisque d'après la sémantique de la hiérarchie, les contraintes d'un niveau donné sont aussi importantes les unes que les autres et on souhaiterait privilégier la satisfaction d'un ensemble de contraintes dont le nombre est supérieur à celui d'un autre ensemble de contraintes du même niveau.

Dans ce chapitre, on reviendra en détail sur ces arguments qui justifient en partie notre démarche qui était de concevoir un nouveau résolveur de propagation locale basé sur un ordre plus fin que celui utilisé par les algorithmes de propagation locale existants. Ce mode de comparaison prend en compte différents modes de combinaison des erreurs par niveau et utilise une agrégation globale de type lexicographique sur les valeurs de ces combinaisons.

Ce résolveur travaille aussi en deux passes: construction d'un graphe (dit graphe-solution) qui minimise (ou maximise) le critère de comparaison utilisé, et exécution des méthodes de ce graphe. Pour construire ce type de graphe nous avons été amenés à utiliser un algorithme de recherche du type "meilleur d'abord" où un noeud représentant un état est un graphe de méthodes ou un ensemble de graphes de méthodes, une branche représente un ajout ou un retrait de contrainte et enfin la fonction d'évaluation dépend du critère global utilisé.

Avec une bonne heuristique, on souhaite par l'utilisation d'un algorithme du type "meilleur d'abord", arriver rapidement à une solution pour le problème qui nous concerne, c'est-à-dire la création d'un graphe solution qui correspond au critère global utilisé. On espère aussi que le nombre de noeuds générés soit plus faible que si on utilisait une recherche en profondeur. L'inconvénient est le coût en mémoire car on stocke certains états.

On intègre deux séries de techniques : la première série a pour objectif de contribuer à réduire le temps de traitement lors de l'examen d'un noeud. Cette série repose sur un mode de représentation des graphes de méthodes que nous avons créé. Ce mode de représentation nous permet de manipuler des graphes de méthodes comme des ensembles de variables. La deuxième série de techniques a pour objectif de contribuer à la réduction du nombre de noeuds à développer lors de la recherche d'une solution. Elle veille à ce que tout noeud créé dans l'arbre de recherche soit maximal (ne contienne pas d'information redondante avec un autre noeud déjà développé).

La fonction d'évaluation dépend du critère utilisé. Cette fonction repose sur les étiquettes des contraintes ajoutées ou retranchées au système. Ceci fait qu'on ne développe un noeud que si on est certain qu'il pourra produire une solution optimale (un graphe solution dit *GLM*).

Ce chapitre est composé de quatre parties : la première partie est une vue assez générale sur ce nouveau résolveur. La deuxième partie décrit la définition du premier critère de comparaison intégré dans cet algorithme ainsi que la définition correspondante du graphe solution. La troisième partie décrit quelques définitions utilisées qui constituent le fondement théorique de la première série de techniques utilisées pour réduire le temps de traitement. Ces définitions sont aussi utilisées pour la planification d'un graphe solution. Dans la quatrième partie, on présentera le descriptif de la deuxième série de techniques qui vise à réduire la complexité en espace mémoire. Cette partie présente également la fonction d'évaluation qui sert à optimiser la recherche et qui permet de déterminer le noeud le plus prometteur dans l'arbre de recherche.

5.1 Vue générale de Houria

Dans le cadre théorique des hiérarchies de contraintes, nous avons vu que la résolution d'une telle hiérarchie équivaut à partitionner l'ensemble des valuations en deux ensembles : un ensemble S_0 qui contient les valuations qui satisfont les contraintes étiquetées par *requise* et un autre qui contient son complémentaire. Ensuite, les valuations dans ce premier ensemble sont ordonnées selon un critère de comparaison intégré dans le résolveur et par conséquent on obtient l'ensemble S (qui contient les solutions à la hiérarchie).

Ici, on se place de plus dans le cadre où les contraintes sont fonctionnelles (chaque contrainte possède une méthode ou plusieurs méthodes qui peuvent être sélectionnées pour la satisfaction de cette contrainte). Le principe décrit auparavant reste le même sauf qu'une facilité supplémentaire s'ajoute pour la résolution d'une telle hiérarchie. Il suffit de construire un graphe de méthodes des contraintes de la hiérarchie sous deux conditions.

La première condition est que ce graphe doit contenir une méthode de chacune des contraintes requises de la hiérarchie. La deuxième condition est que cette construction doit satisfaire le critère intégré dans le résolveur. Lorsqu'un graphe de méthodes est construit en respectant la première condition, on parle de graphe admissible. Lorsqu'un graphe de méthodes est construit en respectant la première et la deuxième conditions, on parle de graphe-solution qui est un *Graphe-Lexicographiquement-Meilleur* (*GLM*[1]). L'exécution dans un ordre topologique des méthodes qui composent un graphe-solution *GLM* produit une solution dans l'ensemble *S*.

Houria est un algorithme incrémental. Il permet le maintien de solutions préférées qui sont dans les noeuds de l'arbre (graphes-solutions) selon le premier critère qui est le nombre de contraintes non satisfaites par niveau, dans une hiérarchie de contraintes fonctionnelles. L'ensemble des noeuds de l'arbre de recherche est dans G. Cet algorithme modifie le noeud courant qui comporte la solution courante ou tente de développer d'autres noeuds afin de trouver une autre solution en fonction des modifications portées sur l'ensemble des contraintes de la hiérarchie. Ces modifications peuvent être soit l'ajout, soit le retrait d'une contrainte.

Houria gère les contraintes dans un ensemble *G* de graphes-admissibles par construction d'un *Graphe-Lexicographiquement-Meilleur* (*GLM*) qui est un graphe-solution. Cet ensemble *G* peut être vu comme l'ensemble des noeuds de l'arbre de recherche. *G* possède la propriété de contenir le meilleur noeud en tête. Ce noeud comporte un ou plusieurs graphes-admissibles qui sont des graphes-solutions (*GLM*). Les autres noeuds dans *G* contiennent des graphes-admissibles. Une fois le graphe-solution *GLM* construit, Houria exécute ses méthodes (dite actives) pour satisfaire les contraintes correspondantes (dite actives)[2].

Les contraintes fonctionnelles traitées par ce système peuvent être des contraintes possédant plusieurs méthodes (multi-directionnelles) et chacune des méthodes peut être une méthode multi-sorties (possédant plusieurs variables de sortie). Le mode de représentation des contraintes utilisé par Houria est souple : il permet de manipuler les graphes-admissibles dans *G* comme des ensembles de variables, ce qui rend les calculs moins coûteux pour la détermination d'une solution à l'issue de l'ajout ou du retrait d'une contrainte.

Deux propriétés essentielles caractérisent les graphes-admissibles dans *G*, la première est qu'aucun de ces graphes ne possède de conflit de méthodes. La deuxième est qu'aucun de ces graphes n'a de méthodes qui forment un circuit.

5.2 Intégration du premier critère de comparaison

5.2.1 Définition formelle du critère utilisé

Le premier critère de comparaison intégré dans cet algorithme est le comparateur *Nombre-Contrainte-Non-Satisfaite* qui cherche l'ensemble des valuations qui satisfont le maximum de contraintes de préférence dans chaque niveau de la hiérarchie. Ce comparateur utilise le nombre de contraintes non-satisfaites dans chaque niveau de la hiérarchie. Il s'agit d'un comparateur global qui discrimine mieux l'ensemble des valuations qui satisfont les contraintes requises qu'un comparateur local. Par conséquent, la taille de l'ensemble des solutions d'une hiérarchie en utilisant ce critère est généralement plus petite que si on utilise le critère *localement-meilleur*. Dans le pire des cas, cette taille est identique. Pour illustrer ceci, considérons l'exemple suivant :

1. Ici, *GLM* designe Graphe Lexicographiquement Meilleur et non Graphe Localement Meilleur utilisé dans le chapitre précédent.
2. une contrainte est active dans le graphe-solution *GLM* si elle possède une méthode active dans ce graphe-solution *GLM*

Exemple :

Soient $H = (H_0, H_1, H_2)$, $H_0 = \{c_{01}, c_{02}\}$, $H_1 = \{c_{11}, c_{12}, c_{13}\}$, $H_2 = \{c_{21}, c_{22}, c_{23}, c_{33}, c_{43}, c_{53}\}$, $S_0 = (\theta, \eta, \zeta)$. On suppose aussi que les séquences d'erreurs obtenues par les valuations θ, η, ζ sur les classes H_0, H_1 et H_2 sont :

$E(H_0\ \theta) = (0,0)$, $E(H_0\ \eta) = (0,0)$, $E(H_0\ \zeta) = (0,0)$,

$E(H_1\ \theta) = (0,0,1)$, $E(H_1\ \eta) = (0,1,0)$, $E(H_1\ \zeta) = (1,0,1)$,

$E(H_2\ \theta) = (0,0,0,0,1)$, $E(H_2\ \eta) = (0,0,0,1,1)$, $E(H_2\ \zeta) = (0,0,0,0,0)$.

Avec le critère de comparaison *Localement-Prédicat-Meilleur* l'ensemble S est (θ, η). La valuation ζ est éliminée à cause de la valuation θ qui satisfait un sur-ensemble de contraintes au niveau H_1. Les valuations θ et η ne peuvent pas être départagées par ce critère de comparaison puisqu'elles satisfont des ensembles non inclus l'un dans l'autre de contraintes au niveau H_1 de la hiérarchie. Tandis qu'avec le critère global *Nombre-Contrainte-Non-Satisfaite*, on jugera au niveau H_1 de la hiérarchie que θ et η sont meilleures que ζ et au niveau H_2 on jugera que la valuation θ est meilleure que η et par conséquent l'ensemble S contient la seule valuation θ.

Définition 5.1:

Une valuation θ est *Globalement-Meilleure* qu'une autre valuation η, si et seulement si, pour chacun des niveaux jusqu'au niveau k-1, le nombre de contraintes non satisfaites après application de θ est égal à celui après application de η, et au niveau k, ce nombre est strictement inférieur.

Nombre-Contrainte-Non-Satisfaite$(\theta, \eta, H) \Leftrightarrow$ *Globalement-Meilleur*(θ, η, H, g),

$$\text{avec} \quad g(V) = \left(\sum_{i=1}^{|V|} v_i \right) \quad (v_i = 0 \text{ si la contrainte est satisfaite et 1 sinon})$$

D'après cette définition, S ne peut pas contenir une valuation jugée par ce critère moins bonne qu'une autre valuation dans S. Cependant, du fait que "*meilleur*" ne crée pas un ordre total, S peut contenir plusieurs solutions telles qu'aucune d'entre elles n'est meilleure que les autres. Si plusieurs solutions existent, notre résolveur donne une de ces solutions, et peut calculer les autres solutions de S si le besoin se présente, contrairement à la technique utilisée dans [Fre88], qui fournit un algorithme basé sur le comparateur *Localement-Prédicat-Meilleur*, et qui donne une seule solution de S. Une autre technique existante permet d'énumérer les solutions de S une par une en effectuant des retours arrière. Cette dernière est fournie par l'interprète de contraintes hiérarchiques dans les langages de programmation logique [BMM+89].

Dans le paragraphe suivant, nous allons présenter comment trouver les valuations de l'ensemble S, en construisant en premier le(s) graphe(s) appelé(s) graphe(s)-solution(s), et en exécutant dans un ordre topologique les méthodes qui composent ce(s) graphe(s).

5.2.2 Graphe Lexicographiquement Meilleur (*GLM*)

Une contrainte est considérée satisfaite si elle est active dans le graphe-solution. Une contrainte est active dans un graphe-admissible si elle possède une méthode incluse dans ce graphe-admissible. Une contrainte est considérée non satisfaite ou non active si elle ne possède aucune méthode incluse dans le graphe-admissible. Un graphe est admissible s'il active toutes les contraintes requises. Le résolveur doit choisir les meilleurs de ces graphes admissibles pour obtenir la (les) meilleure(s) valuation(s) qui satisfait (satisfont) la hiérarchie en utilisant le critère sur lequel il est basé. Dans ce contexte, on parle de planification de graphe.

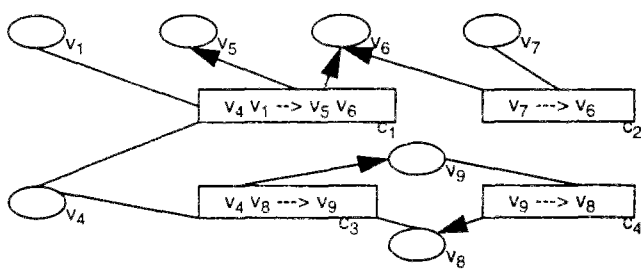
Houria planifie un graphe lexicographiquement meilleur (*GLM*) en utilisant le critère défini dans la section 5.2.1 et applique l'algorithme de propagation locale décrit dans le chapitre précédent pour obtenir les valuations dans S.

Un graphe-solution *GLM* est représenté au moyen d'une ou plusieurs méthodes. Une méthode est une procédure qui lit les valeurs des variables de la partie antécédente de la méthode, et calcule les valeurs des variables conséquentes de cette méthode pour satisfaire la contrainte.

Un graphe-solution *GLM* contient au plus une méthode de chaque contrainte du système. L'algorithme proposé construit le(s) graphe(s)-solution(s) maximal(aux) ne possédant pas de circuits, ni de conflits entre leurs différentes méthodes. Ce(s) graphe(s)-solution(s) est (sont) résolu(s) par la propagation locale.

Si un graphe contient deux méthodes de deux contraintes distinctes, et que ces deux méthodes possèdent au moins une variable commune dans leurs parties conséquentes, alors il y a conflit de méthodes. Par exemple, dans la figure 17, il y a un conflit de méthodes entre la contrainte c_1 et la contrainte c_2. Les deux contraintes ne peuvent être satisfaites simultanément par propagation locale. Si c_1 est satisfaite par l'exécution de sa méthode active, et si ensuite c_2 est satisfaite par l'exécution de sa méthode active, il est très probable que c_1 devienne non satisfaite. Dans cette même figure, il y a aussi existence d'un circuit entre les méthodes des deux contraintes c_3 et c_4. Si c_4 est satisfaite par l'exécution de sa méthode active qui détermine la valeur de la variable v_8 et ensuite la contrainte c_3 est satisfaite par l'exécution de sa méthode active qui détermine la valeur de la variable v_9 alors il est très probable que c_4 devienne non satisfaite. L'algorithme que nous proposons construit des graphes sans conflit de méthodes et sans circuits. Un graphe ainsi construit est utilisé ensuite pour la satisfaction de ses contraintes actives, par exécution dans un ordre topologique de ses méthodes actives (les variables conséquentes des méthodes actives ne sont lues que si leurs valeurs sont calculées avant).

FIGURE 17 : Conflit de méthodes (en v_6), et un circuit direct entre c_3 et c_4



Dans la suite de ce paragraphe, on donnera la définition générale d'un graphe-solution *GLM* et ensuite on donnera une instance de cette définition qui repose sur le critère *Nombre-Contrainte-Non-Satisfaite*.

Définition 5.2

Un graphe-admissible est un graphe-solution, si et seulement si, il ne possède pas de circuit ni de conflit de méthode, et s'il ne possède pas de méthode inactive à un niveau k, telle que si on activait cette méthode, elle générerait un graphe lexicographiquement meilleur.

Définition 5.3

Soient deux graphes-admissibles s_1 et s_2, s_1 est un graphe lexicographiquement meilleur que s_2 si et seulement si, pour tout niveau jusqu'au niveau *k-1*, le nombre de contraintes non actives dans s_1 est égal au nombre de contraintes non actives dans s_2, et au niveau k ce nombre est strictement inférieur.

L'algorithme proposé prend en compte l'étiquette de chaque contrainte pour construire un graphe-solution *GLM* qui soit lexicographiquement préféré à tout autre graphe-admissible qui peut être construit. Lorsque le système des contraintes de la hiérarchie est sur-contraint, Houria laisse les contraintes faibles non satisfaites pour satisfaire une ou plusieurs contraintes étiquetées par des étiquettes considérées comme plus fortes. Par exemple, considérons le graphe-admissible de la figure 18, ce graphe-admissible n'est pas un *GLM*, car si l'on active la méthode (dont l'étiquette est *forte*) de la contrainte c_2 qui détermine les valeurs des variables v_4 et v_6, et si on désactive les deux méthodes (dont les étiquettes sont *moyenne* et *faible*) des contraintes c_3 et c_4, on obtient le graphe-admissible de la figure 19 qui est lexicographiquement meilleur que celui de la figure 18. Le graphe-admissible de la figure 19 n'est pas un *GLM*, du fait que l'on peut activer la méthode de la contrainte c_5, ce qui génère le graphe de la figure 20. Ce dernier est un *GLM*, puisque les méthodes des contraintes inactives, ne peuvent pas générer, si elles sont rendues actives, un graphe-admissible lexicographiquement meilleur.

FIGURE 18 : Un Graphe-admissible non Lexicographiquement Meilleur

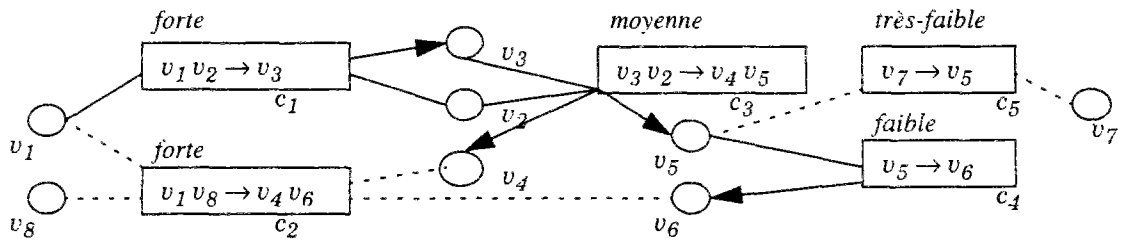


FIGURE 19 : Un Graphe-admissible non Lexicographiquement Meilleur

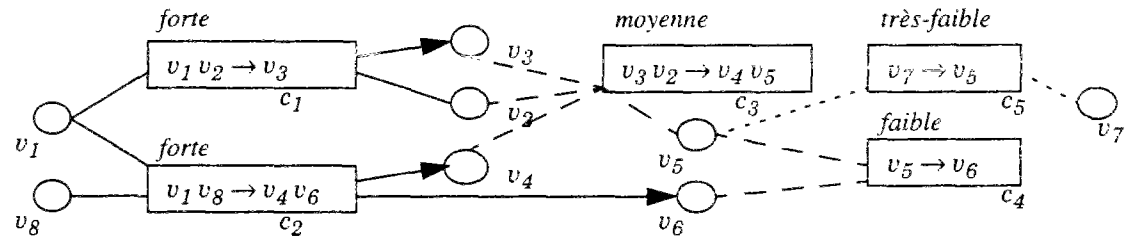
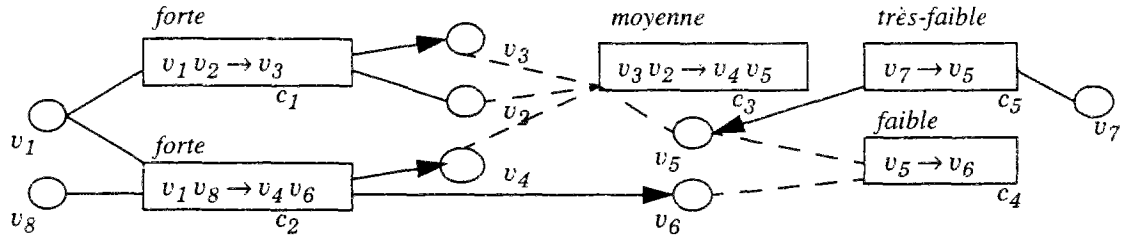


FIGURE 20 : Un Graphe-solution qui est Lexicographiquement Meilleur



5.3 Réduction de temps de traitement et construction d'un *GLM*

Dans cette section, nous allons définir un nouveau mode de représentation des méthodes des contraintes fonctionnelles. Ce nouveau mode de représentation a pour objectif de contribuer à réduire le temps de traitement pour la recherche d'une solution. On définira ensuite la notion de conjonction de ces méthodes pour obtenir la représentation d'un graphe de méthodes. La construction d'un graphe-solution *GLM* est soumise aux différentes conditions citées dans les dernières définitions du paragraphe 5.2.2 à savoir : pas de conflit de méthodes et pas de circuit entre les méthodes. On définira les notions de conflit, de circuit et de consistance entre les méthodes *via* cette nouvelle représentation.

5.3.1 Mode de représentation d'une contrainte fonctionnelle

Notation:

On dénote par $\bar{c}$ l'étiquette de la contrainte c, et par M_c l'ensemble des méthodes de la contrainte c. Chaque méthode m_i dans M_c sera représentée par un couple *(triplet, étiquettes)* où *triplet* est de la forme : *(Ant, Int, Cons)*. Ce triplet servira aussi pour représenter un ensemble de méthodes. *Ant* est l'ensemble des variables antécédentes de la méthode, *Int* est l'ensemble des variables internes, dans le cas d'une seule méthode, cet ensemble est vide. *Cons* est l'ensemble des variables conséquentes de la méthode. Le deuxième élément de ce couple est une liste contenant l'étiquette de la contrainte qui contient cette méthode.

Exemples :

$c \equiv \{v_1 = v_2 - v_3\}$ $\quad \bar{c}$ = (requise)

$M_c \equiv \{(v_2 - v_3 \rightarrow v_1),(v_1 + v_3 \rightarrow v_2),(v_2 - v_1 \rightarrow v_3)\}$

$m_1 \equiv ((\{v_2 v_3\},\{\ \},\{v_1\}),(\text{requise}))$

$m_2 \equiv ((\{v_1 v_3\},\{\ \},\{v_2\}),(\text{requise}))$

$m_3 \equiv ((\{v_1 v_2\},\{\ \},\{v_3\}),(\text{requise}))$.

$c \equiv \{v_2 = v_4\}$ $\quad \bar{c}$ = (forte)

$M_c \equiv \{(v_2 \rightarrow v_4),(v_4 \rightarrow v_2)\}$

$m_1 \equiv ((\{v_2\},\{\ \},\{v_4\}),(\text{forte}))$

$m_2 \equiv ((\{v_4\},\{\ \},\{v_2\}),(\text{forte}))$.

5.3.2 Conjonction d'une méthode et d'un ensemble de méthodes

L'aspect incrémental du résolveur que nous concevons utilise la conjonction entre une nouvelle méthode introduite dans le système et l'ensemble des méthodes existantes qui forment les graphes-admissibles. La prise en compte de cette méthode sur un graphe de méthodes est réalisée par le connecteur de conjonction $\wedge$. En d'autres termes : un graphe de méthodes peut être vu aussi comme un couple *(triplet, étiquettes)*. *triplet* est la représentation des conjonctions des méthodes actives dans le graphe et *étiquettes* est l'ensemble ordonné des étiquettes des méthodes actives dans ce graphe. Pour obtenir ce couple, on définit la conjonction de la représentation d'une méthode et de celle d'un ensemble de méthodes.

Définition 5.4:

Soit la représentation d'une méthode m de la contrainte c : $m \equiv ((Ant(m),\{\ \},Cons(m)),\bar{c})$ et soit la représentation d'un ensemble de méthodes $s \equiv ((Ant(s),Int(s),Cons(s)),\bar{s})$. La représentation de la conjonction de m et de s est :

$$m \wedge s \equiv ((Ant(m \wedge s), Int(m \wedge s), Cons(m \wedge s)), \bar{c} \oplus \bar{s})$$

telles que :

$$Ant(m \wedge s) = (Ant(m) \cup Ant(s)) \backslash (Cons(s) \cup Int(s) \cup Cons(m))$$
$$Int(m \wedge s) = Int(s) \cup (Ant(m) \cap Cons(s)) \cup (Cons(m) \cap Ant(s))$$
$$Cons(m \wedge s) = (Cons(m) \cup Cons(s)) \backslash (Ant(s) \cup Int(s) \cup Ant(m))$$

L'opérateur $\oplus$ concatène deux listes ordonnées d'étiquettes en une liste ordonnée.

La représentation de la conjonction d'une méthode m et d'un ensemble de méthodes s est un couple de la forme *(triplet, étiquettes)*. Il est à noter que l'ensemble *Int* de cette conjonction contient l'ensemble des variables communes trouvées :

dans la partie antécédente de s et dans la partie conséquente de m,

dans la partie conséquente de s et dans la partie antécédente de m,

dans la partie interne de s.

Exemple :

Soit la représentation d'une méthode de la contrainte c, $m \equiv ((\{v_2 v_3\},\{\ \},\{v_1\})$, (forte)) et soit la représentation d'un ensemble[1] de méthodes, $s \equiv ((\{v_4\},\{\ \},\{v_2 v_5\})$, (moyenne , faible)). La représentation de la conjonction est : $m \wedge s \equiv ((\{v_3 v_4\},\{v_2\},\{v_1 v_5\})$,(forte, moyenne, faible)).

Notre motivation pour créer ce mode de représentation est d'être capable d'opérer sur des ensembles de variables au lieu d'utiliser d'autres types de structures de données complexes. On peut tout simplement examiner un ensemble de variables pour savoir si la conjonction d'une méthode avec un ensemble de méthodes constitue un conflit ou un circuit de méthodes. De plus, le fait de distinguer trois ensembles de variables: antécédent, interne et conséquent (*c.à.d* en terme de graphe on parle de source, source et puits, puits) nous permet de répartir l'ensemble des variables de la hiérarchie en des sous-ensembles de tailles relativement petites pour une efficacité de calcul et donc du temps de traitement pour attribuer une sémantique (entrante, sortante ou interne) à chaque variable du graphe.

5.3.3 Soustraction d'une méthode d'un ensemble de méthode

L'aspect incrémental de Houria utilise aussi la soustraction d'une méthode de l'ensemble de méthodes existantes qui forment les graphes. La prise en compte de cette soustraction est réalisée par le connecteur de soustraction — (il est bien évident que ce connecteur peut être défini par l'utilisation de la conjonction de toutes les méthodes de s excepté celle à soustraire. Ici on donne une autre définition à cet opérateur qui a pour objectif d'être beaucoup moins coûteux en calcul lors de son implémentation). Le graphe de méthodes résultant est aussi un couple *(triplet, étiquettes)*. *triplet* est la représentation des conjonctions des méthodes actives excepté la méthode retranchée et *étiquettes* est l'ensemble ordonné des étiquettes de ces méthodes actives dans ce graphe.

1. Dans cet exemple, cet ensemble est réduit à une seule méthode.

Pour obtenir ce couple, on définit d'une façon générale la soustraction de la représentation d'une méthode d'un ensemble de méthodes par la définition suivante : (dans la section 5.3.6 on donnera une version réduite de cette définition, qui sera suffisante pour traiter le problème qui nous intéresse)

Définition 5.5:

Soit la représentation d'une méthode m de la contrainte c : $m \equiv ((Ant(m),\{\ \},Cons(m)),\bar{c})$ et soit la représentation d'un ensemble de méthodes contenant m, $s \equiv ((Ant(s),Int(s),Cons(s)),\bar{s})$. La représentation de la soustraction de m de s est :

$$s - m \equiv ((Ant(s - m), Int(s - m), Cons(s - m)), \bar{s} - \bar{c})$$

telles que :

$$Ant(s - m) = \begin{pmatrix} (Ant(s)\backslash\{v \text{ telle que } v \in Ant(m) \wedge \neg(\exists m' \in s \wedge v \in Ant(m'))\}) \\ \cup \\ \{v \text{ telle que } v \in Cons(m) \cap Int(s) \wedge \neg(\exists m' \in s \wedge v \in Cons(m'))\} \end{pmatrix}$$

$$Int(s - m) = Int(s)\backslash\begin{pmatrix} \{v \text{ telle que } v \in Cons(m) \cap Int(s) \wedge \neg(\exists m' \in s \wedge v \in Cons(m'))\} \\ \cup \\ \{v \text{ telle que } v \in Ant(m) \cap Int(s) \wedge \neg(\exists m' \in s \wedge v \in Ant(m'))\} \end{pmatrix}$$

$$Cons(s - m) = \begin{pmatrix} (Cons(s)\backslash\{v \text{ telle que } v \in Cons(m) \wedge \neg(\exists m' \in s \wedge v \in Cons(m'))\}) \\ \cup \\ \{v \text{ telle que } v \in Ant(m) \cap Int(s) \wedge \neg(\exists m' \in s \wedge v \in Ant(m'))\} \end{pmatrix}$$

Concrètement, cette définition veut dire que :

. La partie antécédente de $s - m$ contient l'union des deux ensembles suivants:

. l'ensemble des variables de la partie antécédente de s excepté celles de la partie antécédente de m qui ne figurent dans aucune partie antécédente d'autres méthodes;

. l'ensemble des variables de la partie conséquente de m qui sont internes dans s et qui ne sont pas dans la partie conséquente d'une autre méthode[1].

. La partie interne de $s - m$ contient l'ensemble des variables de la partie interne de s excepté l'ensemble formé par l'union des deux ensembles suivants :

. l'ensemble des variables dans la partie conséquente de m qui sont internes dans s et qui ne sont pas dans la partie conséquente d'une autre méthode;

. l'ensemble des variables dans la partie antécédente de m qui sont internes dans s et qui ne sont pas dans la partie antécédente d'une autre méthode.

. La partie conséquente de $s - m$ contient l'union des deux ensembles suivants:

. l'ensemble des variables de la partie conséquente de s excepté celles de la partie conséquente de m qui ne figurent dans aucune partie conséquente d'autres méthodes.

. l'ensemble des variables dans la partie antécédente de m qui sont internes dans s et qui ne sont pas dans la partie antécédente d'une autre méthode[2].

. Le deuxième élément du triplet est la liste des étiquettes obtenues par l'application de l'opérateur – qui ôte de la liste d'étiquettes de s l'étiquette de la contrainte c.

1 Une variable dans la partie conséquente de m peut se trouver dans la partie interne de s, dans ce cas cette variable est aussi antécédente d'une autre méthode de s. Cette variable sera retirée de la partie interne de s et mise dans la partie antécédente de $s - m$ uniquement s'il n'existe pas une autre méthode dans s *qui a* dans sa partie conséquente cette variable.

2. Une variable dans la partie antécédente de m peut se trouver dans la partie interne de s, dans ce cas cette variable est aussi conséquente d'une autre méthode. Cette variable sera retirée de la partie interne de s et mise dans la partie conséquente de $s - m$ uniquement s'il n'existe pas une autre méthode dans s *qui a* dans sa partie antécédente cette variable.

5.3.4 Composantes connexes d'un ensemble de méthodes

Pour des raisons d'efficacité d'implémentation, on peut remarquer qu'une représentation d'un ensemble M de méthodes, peut être vue comme l'ensemble des représentations des composantes connexes de M. Cette remarque est formalisée par la définition et les deux propriétés suivantes :

Définition 5.6 :

Soit s la représentation d'un ensemble de méthodes M telle que : $s \equiv ((Ant(s),Int(s),Cons(s)),\bar{s})$ et soit g_i un ensemble de méthodes de M qui forme une composante connexe.

$$g_i = (m_0 \ldots\ldots m_n) \text{ telle que: } \left(\begin{array}{c} \forall m_k (0 \le k \le n) m_k \in M \\ \left(\begin{array}{c} \forall m_i \exists m_j (0 \le i \ne j \le n) \\ \left(\begin{array}{c} Cons(m_i) \cap Ant(m_j) \ne \varnothing \vee Ant(m_i) \cap Cons(m_j) \ne \varnothing \\ \vee \\ Cons(m_i) \cap Cons(m_j) \ne \varnothing \vee Ant(m_i) \cap Ant(m_j) \ne \varnothing \end{array} \right) \end{array} \right) \end{array} \right)$$

Soit C_M l'ensemble des composantes connexes de M.

$$C_M = (g_0 \ldots\ldots g_m) / \forall ij (0 \le i < j \le n)(g_i \cap g_j = \varnothing) \wedge \bigcup_{0 \le k \le m} g_k = M$$

On dénote par C_s l'ensemble des représentations des composantes connexes définies dans C_M.

$$\text{Soit } C_s = \left\{ ((Ant(g_i), Int(g_i), Cons(g_i)),\bar{g_i}) / (g_i \in C_M) \right\}.$$

Propriété 5.1 :

$$(\forall \zeta \eta \in C_s) \left(\begin{array}{c} (Ant(\zeta) \cup Int(\zeta)) \cap (Ant(\eta) \cup Int(\eta)) = \varnothing \\ \wedge \\ (Cons(\zeta) \cup Int(\zeta)) \cap (Cons(\eta) \cup Int(\eta)) = \varnothing \end{array} \right).$$

Propriété 5.2 :

$$s \equiv \left(\left(\bigcup_{g_i \in C_M} Ant(g_i), \bigcup_{g_i \in C_M} Int(g_i), \bigcup_{g_i \in C_M} Cons(g_i) \right), \oplus \bar{g_i} \right).$$

5.3.5 Consistance d'une méthode et d'un ensemble de méthodes

Lors de la conjonction d'une méthode avec un ensemble de méthodes, on est ramené à réaliser le test de la consistance d'une méthode et d'un ensemble de méthodes. Par référence à la définition d'un *GLM* présentée dans la section 5.2.2, la notion de consistance d'une méthode et d'un ensemble de méthodes repose sur les notions de conflit et de circuit. (on distingne les deux notions : circuit et conflit puisque ces deux notions peuvent être indépendantes, on peut avoir un circuit dans un graphe de méthodes sans avoir de conflit et aussi on peut avoir un conflit de méthodes en une ou plusieurs variables dans un graphe de méthodes sans avoir de circuit dans ce graphe). Ces deux notions sont définies via les représentations respectives de la méthode et des composantes connexes de l'ensemble des méthodes par les définitions suivantes :

Définition 5.7:

Soient s la représentation d'un ensemble de méthodes M et C_s l'ensemble des représentations des composantes connexes de s. Soit m la représentation d'une méthode.

$$Pas\text{-}De\text{-}Conflit(s, m)$$
$$\Leftrightarrow$$
$$\left(\begin{array}{c}(\forall \zeta \in C_s) \\ (Cons(\zeta) \cap Cons(m) = \varnothing) \wedge (Int(\zeta) \cap Cons(m) = \varnothing) \wedge (Int(m) \cap Cons(\zeta) = \varnothing)\end{array}\right)$$

On a un conflit entre m et s en une variable si et seulement si cette variable se trouve à la fois dans la partie conséquente de la méthode et dans une des deux parties conséquente ou interne d'une composante connexe de s. La troisième condition dans la définition est toujours vraie puisque la partie interne dans la représentation d'une méthode est nulle. Cette condition est mise uniquement pour avoir la symétrie de la relation *Pas-De-Conflit*. Cette opération au pire des cas possède un coût inférieur à $O(N)$ où N est le nombre de variables dans s.

Définition 5.8:

Soient s la représentation d'un ensemble de méthodes M et C_s l'ensemble des représentations des composantes connexes de s. Soit m la représentation d'une méthode.

$$Pas\text{-}De\text{-}Circuit(s, m)$$
$$\Leftrightarrow$$
$$\left(\begin{array}{c}(\forall \zeta \in C_s) \\ \left(\begin{array}{c}\forall v_i \in ((Ant(\zeta) \cup Int(\zeta)) \cap (Cons(m) \cup Int(m))) \wedge \forall v_j \in ((Cons(\zeta) \cup Int(\zeta)) \cap (Ant(m) \cup Int(m))) \\ \Rightarrow \\ \neg Chemin(v_i, v_j)\end{array}\right)\end{array}\right)$$

Théorème 5.1:

Soient s la représentation d'un ensemble de méthodes M et C_s l'ensemble des représentations des composantes connexes de s. Soit m la représentation d'une méthode.

$$(\forall \zeta \in C_s)$$
$$\neg(((Ant(\zeta) \cup Int(\zeta)) \cap (Cons(m) \cup Int(m)) \neq \varnothing) \wedge ((Cons(\zeta) \cup Int(\zeta)) \cap (Ant(m) \cup Int(m)) \neq \varnothing))$$
$$\Rightarrow$$
$$Pas\text{-}De\text{-}Circuit(s, m)$$

Preuve :

Si la condition de l'implication est vraie alors l'un des ensemble dans l'une des expressions dans cette condition (après l'avoir calculé) est vide et par conséquent aucun chemin ne peut exister.

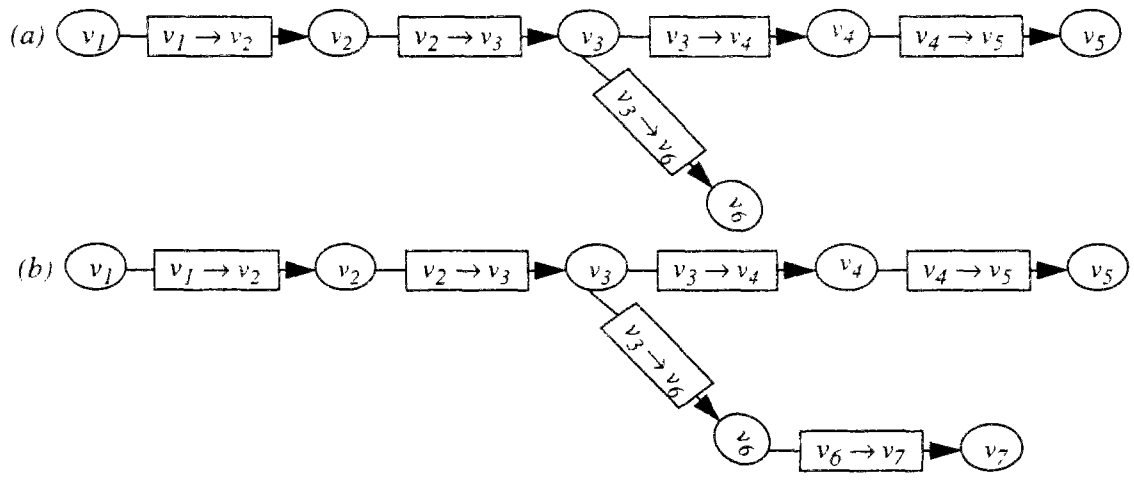
La condition de l'implication dans le théorème 5.1 implique aussi la condition de la partie droite de l'équivalence dans la définition 5.8. La véracité de l'absence d'un circuit entre m et s passe en premier par le test de la condition dans le théorème 5.1. Si l'une des expressions de cette condition est fausse alors on garantit qu'il n'existe pas de circuit entre m et s. Si cette condition est fausse alors la véracité de la relation *Pas-De-Circuit* repose sur la condition de la définition.

Dans ce cas, pour détecter l'existence d'un circuit, il suffit de trouver un chemin *via* une composante de *s* qui a respectivement pour source et puits une variable dans la partie antécédente de *m* et une variable dans la partie conséquente de *m*. Dans ces mêmes conditions, il est bien évident qu'il faut tester toutes les combinaisons entre les variables des deux parties antécédente et conséquente de *m* pour conclure qu'il n'existe pas de circuit entre *s* et *m*. Cependant le coût de cette opération dans le pire des cas reste linéaire puisque le nombre de combinaisons à tester est une constante (pour une méthode possédant 2 variables dans sa partie antécédente et 2 variables dans sa partie conséquente on aura à tester l'existence de 4 chemins) et puisqu'un algorithme de recherche de chemin avec marquage dans un graphe orienté est en O(*N*) (où *N* est le nombre de variables dans le graphe).

Dans la suite de cette section, on donnera deux exemples pour illustrer la définition 5.8 et montrer l'utilisation du théorème 5.1 dans cette définition. Le premier exemple porte sur des contraintes binaires et montre que la véracité de la condition de ce théorème prédit uniquement l'absence de circuit lors d'une conjonction. Le deuxième exemple porte sur des contraintes n-aires. On montre par cet exemple que ce théorème est une heuristique qui optimise la recherche. On montre aussi que le pire cas qui puisse se produire est la fausseté de la condition de l'implication du théorème 5.1 et la véracité de la partie droite de la définition 5.8.

Le premier exemple est celui de la figure 21 qui montre une situation de conjonction d'un ensemble de méthodes *s* et d'une méthode *m*. La figure 21.a montre un graphe de méthodes *s* dont les parties antécédente, interne et conséquente sont respectivement : $\{v_1\}$, $\{v_2,v_3,v_4\}$, $\{v_5,v_6\}$. La figure 21.b montre la conjonction d'une méthode *m* ayant respectivement pour parties antécédente et conséquente les ensembles $\{v_6\}$, $\{v_7\}$ et de l'ensemble de méthodes *s*. La condition dans le théorème est vraie puisque la variable v_7 n'est ni dans l'antécédent de *s* ni dans l'interne de *s* et donc il est inutile d'examiner la partie droite de la définition. Par conséquent, le prédicat *Pas-De-Circuit(m,s)* est vrai. Les figures 21.c et 21.d montrent respectivement la conjonction d'une méthode $m \equiv (\{v_6\}, \{\}, \{v_1\}, \overline{m})$ et de *s* et la conjonction d'une méthode $m \equiv (\{v_2\}, \{\}, \{v_4\}, \overline{m})$ et de *s*. Pour ces deux conjonctions, la condition du théorème est fausse. Ici, il s'agit d'une situation où deux méthodes différentes ont le même comportement sur la condition du théorème et un comportement différent sur la condition dans la partie droite de la définition (puisque dans la figure 21.c on détecte la présence d'un circuit et dans la figure 21.d on garantit l'absence de circuit). Il est donc nécessaire d'examiner la condition de la partie droite de la définition pour prédire l'existence ou l'absence d'un circuit (nous verrons par la suite comment réduire cette définition de sorte à n'examiner que la condition du théorème pour prédire l'existence ou l'absence de circuit lorsqu'il s'agit de contraintes binaires).

FIGURE 21 : Conjonction de méthode avec un ensemble de méthodes et circuit



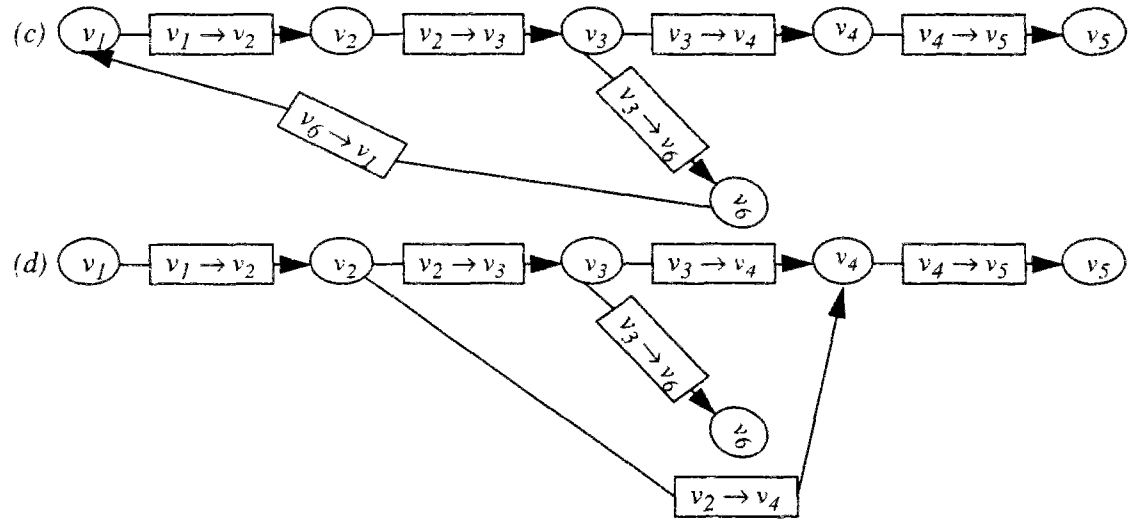
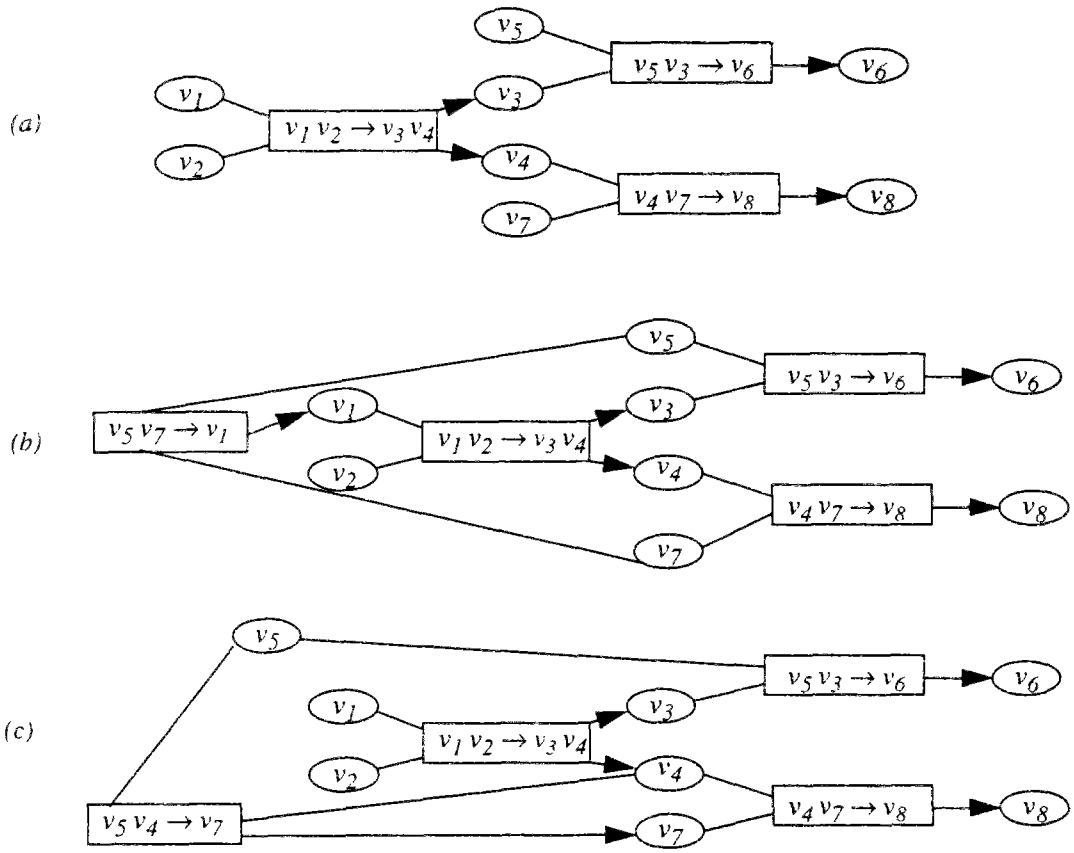


FIGURE 22 : Conjonction de méthode et test de circuit



Le deuxième exemple est celui de la figure 22 qui montre une situation de conjonction d'un ensemble de méthodes *s* et d'une méthode *m*. La figure 22.a montre un graphe *s* de méthodes n-aires dont les parties antécédente, interne et conséquente sont respectivement : $\{v_1,v_2,v_5,v_7\}$, $\{v_3,v_4\}$, $\{v_6,v_8\}$. La figure 22.b montre la conjonction de l'ensemble de méthodes *s* et d'une méthode *m* n-aire ayant respectivement pour parties antécédente et conséquente les ensembles $\{v_5,v_7\}$, $\{v_1\}$.

La condition du théorème est vraie et donc on garantit l'absence de circuit. La figure 22.c montre la conjonction de l'ensemble de méthodes s et d'une méthode m n-aire ayant respectivement pour partie antécédente et conséquente les ensembles $\{v_5,v_4\}$, $\{v_7\}$. La condition du théorème est fausse puisque les deux expressions qui la composent sont vraies (la variable v_7 est dans l'antécédent de s et aussi dans le conséquent de m, et la variable v_4 est dans l'interne de s et aussi dans l'antécédent de m). L'examen de la partie droite de la définition est donc nécessaire maintenant. Cet examen montre l'absence de circuit dans cette conjonction (puisqu'il n'y a pas de chemin qui relie les variables dans l'une des paires (v_7,v_4), (v_7,v_5). Par conséquent, le prédicat *Pas-De-Circuit(m,s)* est vrai.

Lors d'une conjonction d'une méthode m à un ensemble de méthodes s, si on n'a pas de conflit entre m et s (c.à.d. le prédicat *Pas-De-Conflit(m,s)* est vrai), alors la relation *Pas-De-Circuit(m,s)* définie précédemment peut être réduite tout en gardant sa complétude. Cette réduction est faite de sorte à ne pas refaire les opérations déjà effectuées pour déterminer la valeur logique du prédicat *Pas-De-Conflit(m,s)*. La définition suivante montre la relation *Pas-De-Circuit-bis* qui est une version réduite de la définition précédente. Cette définition est suffisante pour prédire l'existence ou l'absence d'un circuit à la suite d'une conjonction lorsque le prédicat *Pas-De-Conflit* est vrai. L'absence d'un circuit est garantie par la véracité de la première condition de l'implication du théorème 5.2.

Définition 5.9:

Soient s la représentation d'un ensemble de méthodes M et C_s l'ensemble des représentations des composantes connexes de s. Soit m la représentation d'une méthode.

$$Pas\text{-}De\text{-}Circuit\text{-}bis(s, m)$$
$$\Leftrightarrow$$
$$\left(\begin{array}{c}(\forall \zeta \in C_s) \\ \forall v_i \in ((Cons(\zeta) \cup Int(\zeta)) \cap Ant(m)) \wedge \forall v_j \in (Ant(\zeta) \cap (Cons(m) \cup Int(m))) \\ \Rightarrow \\ \neg Chemin(v_i, v_j)\end{array}\right)$$

Théorème 5.2 :

Soient s la représentation d'un ensemble de méthodes M et C_s l'ensemble des représentations des composantes connexes de s. Soit m la représentation d'une méthode.

$$(\forall \zeta \in C_s)$$
$$\neg((Ant(\zeta) \cap (Cons(m) \cup Int(m)) \neq \varnothing) \wedge ((Cons(\zeta) \cup Int(\zeta)) \cap Ant(m) \neq \varnothing))$$
$$\Rightarrow$$
$$Pas\text{-}De\text{-}Circuit\text{-}bis(s, m)$$

Preuve: identique à celle du théorème 5.1.

Théorème 5.3 :

Si les contraintes du système sont des contraintes binaires alors la première condition de l'implication du théorème 5.2 est une condition suffisante pour déterminer la véracité de l'existence ou de l'absence d'un circuit entre s et m *(c.à.d* la condition est équivalente au prédicat *Pas-De-Circuit-bis).*

Preuve :

Si la première condition est vraie, alors par définition on peut déduire l'absence de circuit. Si la première condition est fausse, ceci veut dire que la variable dans la partie antécédente de m est aussi dans l'une des deux parties interne ou conséquente d'une composante connexe ξ de s. De plus la

variable dans la partie conséquente de m ne peut se trouver que dans la partie antécédente de ξ puisque sinon on aurait eu un conflit entre m et s. Maintenant, puisqu'il s'agit de contraintes binaires, si on considére la variable de la partie antécédente de ξ comme un puits et celle dans l'une des parties interne ou conséquente de ξ comme une source et de les relier par m créera forcément un circuit.

La consistance d'une méthode m et d'un ensemble de méthodes s est basée sur les définitions précédentes. La définition suivante considère ce fait :

Définition 5.10:

Soient s la représentation d'un ensemble de méthodes M et m la représentation d'une méthode. On dira que s est consistante avec m si et seulement si on a ni conflit ni circuit entre m et s.

$$Consistant(s, m) \Leftrightarrow Pas\text{-}De\text{-}Conflit(s, m) \wedge Pas\text{-}De\text{-}Circiut\text{-}Bis(s, m)$$

5.3.6 Soustraction d'une méthode d'un ensemble de méthodes non conflictuelles

Après avoir défini la notion de *Pas-De-Conflit*, la définition de la soustraction d'une méthode m d'un ensemble de méthodes s présenté à la section 5.3.4 peut être réduite puisque notre objectif est de soustraire une méthode d'un ensemble de méthodes non conflictuelles (*c.à.d.* il n'existe pas deux méthodes dans s qui possèdent une variable commune dans leurs parties conséquentes). Soit la nouvelle version de la définition de l'opérateur — :

Définition 5.11:

Soient s la représentation d'un ensemble de méthodes M et C_s l'ensemble des représentations des composantes connexes de s. Soit ζ la composante connexe dans C_s contenant la méthode m de la contrainte c à soustraire. La représentation de la soustraction de m de ζ est :

$$\zeta - m \equiv ((Ant(\zeta - m), Int(\zeta - m), Cons(\zeta - m)), \bar{s} - \bar{c})$$

telles que :

$$Ant(\zeta - m) = \begin{pmatrix} (Ant(\zeta) \backslash \{v \text{ telle que } v \in Ant(m) \wedge \neg(\exists m' \in \zeta \wedge v \in Ant(m'))\}) \\ \cup \\ (Cons(m) \cap Int(\zeta)) \end{pmatrix}$$

$$Int(\zeta - m) = \begin{pmatrix} Int(\zeta) \backslash \\ (Cons(m) \cup \{v \text{ telle que } v \in Ant(m) \cap Int(\zeta) \wedge \neg(\exists m' \in s \wedge v \in Ant(m'))\}) \end{pmatrix}$$

$$Cons(\zeta - m) = \begin{pmatrix} (Cons(\zeta) \backslash Cons(m)) \\ \cup \\ \{v \text{ telle que } v \in Ant(m) \cap Int(\zeta) \wedge \neg(\exists m' \in s \wedge v \in Ant(m'))\} \end{pmatrix}$$

La différence par rapport à l'ancienne version de cette définition est qu'ici, puisqu'on suppose que l'ensemble s ne contient pas de méthodes conflictuelles et donc ξ non plus, il est inutile de considérer l'existence d'une méthode possédant en sa partie conséquente une variable se trouvant aussi dans la partie conséquente de m. Et donc, si la partie conséquente de m contient une variable se trouvant dans la partie interne de ξ alors elle est ôtée de la partie interne de $\xi - m$ et mise automatiquement dans la partie antécédente de $\xi - m$ (il est à noter ici que cette opération de soustraction peut conduire à la création de plusieurs composantes connexes).

5.4 Fonction d'évaluation et réduction de l'espace mémoire

Cette section décrit l'approche algorithmique utilisée dans Houria. Cette approche est basée sur :

- la définition d'un *GLM* décrite dans la section 5.2 (à savoir : pas de conflit ni de circuit et le nombre de contraintes actives par niveau est maximum)

- les outils définis dans la section 5.3. (à savoir : la représentation d'une méthode et d'un ensemble de méthodes, les composantes connexes, les opérateurs $\wedge$, $\oplus$ et — , *Pas-De-Conflit*, *Pas-De-Circuit-bis* et enfin les deux théorèmes 5.2 et 5.3)

- une technique d'optimisation qui contribue à réduire la complexité en espace mémoire (cette technique est presque obligatoire)

- la fonction d'évaluation qui sert à optimiser la recherche et qui permet de déterminer le meilleur nœud dans l'arbre de recherche à développer.

Nous rappelons qu'un noeud dans G représente un état qui est un graphe de méthodes ou un ensemble de graphes de méthodes, une branche représente un ajout ou un retrait de contrainte et enfin la fonction d'évaluation va dépendre du critère global utilisé.

Dans un premier temps, nous allons présenter dans la section 4.1 une approche pour déterminer les éléments de G lors de l'ajout ou du retrait d'une contrainte. La section 4.2 présente une amélioration de cette approche. Cette amélioration consiste en une technique d'optimisation presque obligatoire qui contribue à réduire la complexité en espace mémoire. La section 4.3 présente *via* un exemple la fonction d'évaluation ainsi que l'approche globale utilisé intégrant cette fonction d'évaluation.

5.4.1 Calcul de l'ensemble *G*

L'ensemble G est un ensemble de couples *(ga, ep)* ou *ga* est un graphe admissible et *ep* est une liste ordonnée d'étiquettes. Chaque graphe admissible est représenté par le *triplet (Ant, Int, Cons)* obtenu par la conjonction des méthodes actives dans le graphe. La liste *ep* d'étiquettes est composée par l'ensemble des étiquettes des méthodes actives dans ce graphe admissible. Cette liste est ordonnée selon l'importance des étiquettes qui la composent (elle peut contenir des doublons).

Notre objectif est de déterminer le(s) graphe(s)-solution(s) *GLM* après l'ajout ou le retrait d'une contrainte c à la hiérarchie. Les pas suivants décrivent une première approche pour réaliser ceci:

`S'il s'agit de l'ajout de la contrainte c à la hiérarchie:`

- La contrainte c est ajoutée à la hiérarchie.

- Pour chaque couple dans G, il faut tester si le graphe-admissible dans ce couple est consistant avec une méthode de la contrainte c. Dans le cas où il y a consistance[1], on ajoute à G le nouveau couple formé par : (graphe-admissible $\wedge$ méthode, liste d'étiquettes du graphe-admissible augmenté de l'étiquette de la méthode).

- Retourner le(s) couple(s) dans G ayant la liste d'étiquettes maximale (c'est-à-dire des *GLM*, les couples retournés possèdent des listes d'étiquettes identiques, et de plus elles sont lexicographiquement supérieures aux autres listes des couples restants dans G).

1. Ici la notion de consistance est celle que nous avons définie à la section précédente.

Etant donnée cette approche[1], dans le cas d'ajout d'une contrainte c, il est nécessaire que l'algorithme tente d'ajouter les méthodes de la contrainte c à chaque élément de G, et non seulement aux éléments qui possèdent la liste d'étiquettes la plus élevée (les *GLM*), car, du fait qu'on ne connaît pas les contraintes à venir, on risque de ne pas construire le meilleur graphe-admissible (c.à.d *GLM*).

Exemple :

On suppose que G contient le couple vide (c.à.d. $G = \{(\{\ \},\{\ \},\{\ \}),\ ())\}$) et que l'on a à ajouter successivement les trois méthodes m_1, m_2 et m_3 respectives des trois contraintes différentes c_1, c_2 et c_3 telles que :

$$m_1 \equiv ((\{v_1\},\{\ \},\{v_2\}),\ (\text{forte}))\,,\ m_2 \equiv ((\{v_3\},\{\ \},\{v_2\}),\ (\text{faible}))\,,$$
$$m_3 \equiv ((\{v_1\},\{\ \},\{v_3\}),\ (\text{forte}))\,.$$

Après conjonction de m_1 et du couple vide dans G on a :

$$G = \left\{((\{v_1\},\{\ \},\{v_2\}),\ (\text{forte}))\right\}.$$

A l'ajout de m_2 à l'ensemble G, on observe que m_2 et le couple dans G ne sont pas consistants (puisque le prédicat *Pas-De-Conflit* est faux) Par conséquent, le seul couple est celui formé par la méthode m_2. Ce couple est ajouté à G:

$$G = \left\{((\{v_1\},\{\ \},\{v_2\}),\ (\text{forte})),((\{v_3\},\{\ \},\{v_2\}),\ (\text{faible}))\right\}$$

A l'ajout de m_3 à l'ensemble G, on observe que m_3 est consistante avec les deux couples dans G. Par conséquent deux nouveaux couples sont formés et ajoutés à G. Aussi un nouveau couple est formé (noeud) en considérant la méthode m_3 et ajouté à G (ceci pour garantir la complétude).

$$G = \left\{\begin{array}{c}((\{v_1\},\{\ \},\{v_2v_3\}),\ (\text{forte, forte})),((\{v_1\},\{v_3\},\{v_2\}),\ (\text{forte, faible}))\\ ((\{v_1\},\{\ \},\{v_2\}),\ (\text{forte})),((\{v_1\},\{\ \},\{v_3\}),\ (\text{forte}))\\ ((\{v_3\},\{\ \},\{v_2\}),\ (\text{faible}))\end{array}\right\}$$

Le graphe-solution *GLM* retourné ici est celui possédant la liste des étiquettes maximales, à savoir (*forte, forte*).

S'il s'agit du retrait de la contrainte c[2] de la hierarchie:

- Pour chaque couple dans G, il faut tester si le graphe-admissible dans ce couple contient une des méthodes de la contrainte c. Si c'est le cas alors ce couple est enlevé de G et un nouveau couple est formé par (graphe-admissible — méthode, liste d'étiquettes du graphe admissible diminué de l'étiquette de la méthode). Ce nouveau couple est mis dans S s'il n'existe pas déjà.

- La contrainte c est retirée de la hiérarchie.

- Retourner le(s) couple(s) dans G ayant la liste d'étiquettes maximale.

Dans le cas du retrait de la contrainte c, il est nécessaire d'examiner si une des méthodes de cette contrainte est dans les graphes-admissibles des différents couples dans G. Si une telle méthode existe alors le couple contenant ce graphe-admissible est ôté de G. Un nouveau couple est formé en considérant le couple ôté. Le graphe-admissible du nouveau couple est obtenu par application de l'opérateur — (défini auparavant dans la section 5.3.6) sur la représentation du graphe-admissible et celle de la méthode et par enlèvement de l'étiquette de la méthode de la liste des étiquettes du graphe admissible. Ce nouveau couple obtenu est ajouté à l'ensemble G s'il n'existe pas déjà.

1. On peut considérer que cette approche est basée sur le principe d'une recherche en largeur d'abord.
2. On suppose que le retrait des contraintes ne concerne que les contraintes de préférences et non les contraintes requises.

Exemple :

On reprend l'exemple précédent et on suppose que l'on veut retirer la méthode m_2 des couples dans G. Seuls le deuxième et le cinquième couple dans G contiennent la méthode m_2. Ces deux couples sont ôtés de G. Ensuite, par application de l'opérateur — sur le deuxième couple ôté de G et sur la méthode m_2, on forme le couple *(({v1},{},{v2}),(forte))*, ce couple existe déjà dans G et donc il n'y a pas besoin de le rajouter. Le cinquième couple ôté de S est la méthode elle-même à retirer et donc il n'y a rien à ajouter à G. Par conséquent l'ensemble S après cette opération est :

$$G = \left\{ \begin{array}{c} (\{\{v_1\},\{\ \},\{v_2 v_3\}),\ (\text{forte, forte})),\ (\{\{v_1\},\{\ \},\{v_2\}),\ (\text{forte})) \\ ,(\{\{v_1\},\{\ \},\{v_3\}),\ (\text{forte})) \end{array} \right\}$$

Le graphe-solution *GLM* retourné ici est celui possédant la liste des étiquettes maximales : *(forte, forte)*.

L'approche que nous venons de décrire présente un intérêt au point de vue de la complétude, c'est-à-dire qu'après l'ajout ou le retrait de n contraintes, on est sûr que des meilleurs graphes-admissibles ont été construits et se trouvent dans G.

Cependant, cette approche développe des graphes-admissibles contenant des informations redondantes et de plus elle risque de développer un nombre exponentiel de graphes-admissibles (pour n méthodes on risque d'avoir 2^n graphes-admissibles). Dans un souci d'améliorer cette approche, dans la section suivante, on présentera une technique d'optimisation qui ne développera que des graphes-admissibles n'ayant pas d'informations redondantes. Ceci contribuera à réduire le nombre de graphes-admissibles dans G (c.à.d le nombre de noeud à développer dans l'arbre) et par conséquent l'espace mémoire utilisé.

5.4.2 Réduction de la taille de *G*

Une première amélioration de l'approche décrite ci-dessus consiste à ne garder que les couples contenant les graphes-admissibles maximaux dans G, c'est-à-dire que si lors de la tentative d'ajout d'une méthode à un couple de G, cette méthode est consistante avec le graphe-admissible de ce couple alors la méthode sera ajoutée. Dans le cas contraire, c'est-à-dire quand la méthode est inconsistante avec le graphe-admissible de ce couple, on extrait de ce couple un ensemble de couples maximaux (c'est-à-dire que pour chaque couple de cet ensemble, il n'existe pas un autre couple dans ce même ensemble qui l'inclut).

Le graphe-admissible d'un couple maximal extrait doit être consistant avec la méthode à ajouter. Dans le cas où la méthode à ajouter est inconsistante avec toutes les méthodes du graphe-admissible, alors le couple extrait est le couple *(({}{}{}),())*. Dans ces conditions, le couple qui sera formulé considérera uniquement la méthode à ajouter. Houria construit un ensemble de nouveaux couples en intégrant la méthode à ajouter à chaque couple de l'ensemble des couples maximaux extraits. Chaque nouveau couple construit sera ajouté dans G, dans le cas où il n'existe pas dans G un autre couple qui le recouvre. Cette amélioration garde la complétude de l'algorithme et présente l'avantage de réduire la taille de G.

Il est à noter ici que si l'inconsistance de la méthode et du graphe-admissible du couple est dûe à la fausseté du prédicat *Pas-De-Conflit* alors l'ensemble de couples extraits comporte un seul couple (voir l'exemple ci-dessous). Tandis que si l'inconsistance est due à la fausseté du prédicat *Pas-De-Circuit-bis* et si les contraintes sont binaires, alors le nombre de couples extraits est égal au nombre de méthodes qui forment un circuit avec la méthode à ajouter (puisque dans ce cas pour ne pas avoir de circuit, il suffit d'enlever une des méthodes de cet ensemble et donc le nombre de possibilités est égal au nombre de méthodes dans cet ensemble).

Exemple :

On reprend l'exemple précédent et on suppose que l'on veut ajouter successivement les trois méthodes m_1, m_3 et m_2 .

Après ajout de m_1 on obtient:

$$G = \{((\{v_1\},\{\ \},\{v_2\}),\ (\text{forte}))\}.$$

A l'ajout de m_3 à l'ensemble G, on observe que la méthode m_3 et le couple dans G sont consistants (puisque le prédicat *Pas-De-Conflit* et le prédicat *Pas-De-Circuit-bis* sont vrais) et par conséquent, une mise à jour de ce couple est effectuée et l'ensemble G est maintenant :

$$G = \{((\{v_1\},\{\ \},\{v_2 v_3\}),\ (\text{forte, forte}))\}$$

A l'ajout de m_2 à l'ensemble G, on observe que m_2 est inconsistante avec le couple dans G. Cette inconsistance provient de la fausseté du prédicat *Pas-De-Conflit* puisque la variable v_2 est commune à la partie conséquente du graphe-admissible et à celle de m_2. Par conséquent l'ensemble des couples extraits comporte un seul couple qui est : *(({v1},{},{v3}),(forte))*. Le nouveau couple formé par la conjonction de ce couple extrait et de m_2 est placé dans G puisqu'il est maximal.

$$G = \{((\{v_1\},\{\ \},\{v_2 v_3\}),(\text{forte,forte})),((\{v_1\},\{v_3\},\{v_2\}),\ (\text{forte, faible}))\}.$$

Le graphe-solution *GLM* retourné ici est celui possédant la liste des étiquettes maximale : (*forte, forte*).

En considérant cette amélioration, les pas décrits précédemment concernant le retrait d'une contrainte restent valides à l'exception du fait qu'au pas 1 de la procédure, au lieu de tester l'existence du nouveau couple formulé dans G, il faut tester s'il n'existe pas un couple dans G qui recouvre le nouveau couple formé (ce test a pour but de ne conserver dans G que des couples maximaux). Dans ce cas, le nouveau couple formé sera mis dans G.

Exemple :

Si l'on considère l'exemple précédent et si on retranche la méthode m_2 en utilisant l'opérateur — du deuxième couple dans G, alors le couple obtenu est : *(({v_1},{},{v_3}),(forte))*. Ce couple ne sera pas mis dans G puisqu'il est recouvert par le couple existant dans G : *(({v_1},{},{v_2 v_3}),(forte,forte))*.

5.4.3 Fonction d'évaluation et approche globale

La fonction d'évaluation utilisée par notre résolveur considère la liste des étiquettes des couples dans G. Cette fonction a pour but de diriger le mieux possible la recherche dans l'arbre pour obtenir un graphe-solution *GLM* à la suite d'ajout ou de retrait d'une contrainte. Le but de cette est aussi de retarder le plus possible des calculs inutiles. Nous allons illustrer cette fonction au travers de l'exemple suivant :

Soit $H = \{H_0, H_1, H_2\}$ avec H_0 la classe des contraintes absolues et H_1 et H_2 les classes des contraintes flexibles. H_1 et H_2 sont les classes des contraintes étiquetées respectivement par les étiquettes *forte* et *moyenne*. H_0 contient deux contraintes c_{01} et c_{02} telles que :

$M_{c_{01}} \equiv \{v_1 \rightarrow v_2, v_2 \rightarrow v_1\}$ et $M_{c_{02}} \equiv \{v_2 v_3 \rightarrow v_4\}$, et donc : $G = \{gaep_1^0, gaep_2^0\}$ avec

$gaep_1^0 = ((\{v_1 v_3\}, \{v_2\}, \{v_4\}), (requise, requise))$ et

$gaep_2^0 = ((\{v_2 v_3\}, \{\ \}, \{v_1 v_4\}), (requise, requise))$.

On introduit maintenant dans la classe H_1 la contrainte c_{11} telle que : $M_{c_{11}} \equiv \{v_5 \rightarrow v_1\}$. G est mis à jour et son contenu est le suivant :

$G = \{gaep_1^1, gaep_2^1\}$, $gaep_1^1 = ((\{v_5 v_3\}, \{v_2 v_1\}, \{v_4\}), (requise, requise, forte))$

$gaep_2^1 = ((\{v_2 v_3\}, \{\ \}, \{v_1 v_4\}), (requise, requise))$.

$gaep_2^0 = gaep_2^1$ puisque la méthode de la contrainte c_{11} est inconsistante avec une des contraintes requises dans $gaep_2^0$.

On introduit maintenant dans la classe H_2 la contrainte c_{21} telle que : $M_{c_{21}} \equiv \{v_6 v_7 \rightarrow v_2\}$. G est mis à jour et son contenu est :

$G = \{gaep_1^2, gaep_2^2\}$, $gaep_1^2 = ((\{v_5 v_3\}, \{v_2 v_1\}, \{v_4\}), (requise, requise, forte))$

$gaep_2^2 = ((\{v_6 v_7 v_3\}, \{v_2\}, \{v_1 v_4\}), (requise, requise, moyenne))$.

$gaep_1^1 = gaep_1^2$ puisque la méthode de la contrainte c_{21} est non consistante avec une des méthodes requises dans $gaep_1^1$.

On observe que lorsque la contrainte c_{21} est ajoutée au système, le graphe *GLM* est le graphe admissible dans $gaep_1^1$ et il n'est pas nécessaire de calculer $gaep_2^2$ (puisque *(requise, requise, forte)* $>_{lex}$ *(requise, requise, moyenne)*).

Maintenant, on introduit dans la classe H_2 la contrainte c_{22} telle que $M_{c_{22}} \equiv \{v_7 \rightarrow v_5\}$. G est mis à jour et son contenu est :

$S = \{gaep_1^3, gaep_2^3\}$,

$gaep_1^3 = ((\{v_3 v_7\}, \{v_2 v_1 v_5\}, \{v_4\}), (requise, requise, forte, moyenne))$,

$gaep_2^3 = ((\{v_6 v_7 v_3\}, \{v_2\}, \{v_1 v_4 v_5\}), (requise, requise, moyenne))$

On observe que lorsque c_{22} est ajoutée, le graphe admissible dans le couple $gaep_1^2$ est consistant avec la méthode de la contrainte c_{22}. Puisque la liste d'étiquettes *ep* du couple $gaep_1^2$ est la liste maximale de toutes les listes d'étiquettes dans G, il aurait été suffisant de calculer seulement le couple $gaep_1^3$ puisqu'il contient le graphe *GLM*.

On peut généraliser ces différentes observations et formuler une fonction ($f = g + h$) d'évaluation de la manière suivante:

> l'ensemble G est partitionné en sous-ensembles de couples[1]. Chaque sous-ensemble possède les couples ayant la même liste d'étiquettes et une file d'attente appelée "file des contraintes à ajouter".
>
> chaque sous-ensemble aura deux listes d'étiquettes :
>
> > une liste réelle d'étiquettes (g = liste-réelle-étiquettes) et
> >
> > une liste potentielle d'étiquettes (f = liste-potentielle-étiquettes).
>
> la liste réelle sera la liste d'un des couples du sous-ensemble. La liste potentielle sera la liste réelle du sous-ensemble (c.à.d g) augmentée de la liste d'étiquettes de la file d'attente associée à ce sous-ensemble (c.à.d h). La liste d'une file d'attente comporte les étiquettes des contraintes se trouvant dans cette file d'attente. Les sous-ensembles seront ordonnés dans G selon l'ordre lexicographique de leur liste potentielle d'étiquettes (c.à.d selon f). Le premier sous-ensemble dans G aura toujours la liste potentielle égale à la liste réelle ($f = g$).

Lors de l'ajout d'une contrainte au système, cette contrainte sera ajoutée d'une manière effective[2] au premier sous-ensemble de G, et elle sera mise dans les files d'attentes des autres sous-ensembles. Les sous-ensembles déduits de cette tentative d'ajout effectif sont d'abord utilisés pour ôter de G les couples qui sont recouverts par un des couples dans ces sous-ensembles (ceci pour garder la maximalité des couples dans les sous-ensemble de G). Ensuite, ces sous-ensembles déduits seront ordonnés dans l'ensemble G selon leurs listes-potentielles-étiquettes respectives. La procédure d'ajout effectif s'arrête dans le cas où la queue des contraintes à ajouter du premier sous-ensemble de G est vide. Dans le cas opposé, la procédure tente d'ajouter toutes les contraintes de la queue d'attente du premier sous-ensemble de G, aux différents couples de ce sous-ensemble et aux couples résultants de cet ajout. Ceci est répété jusqu'à ce que la liste potentielle d'étiquettes du premier sous-ensemble ne soit pas lexicographiquement strictement supérieure à la liste réelle du premier sous-ensemble.

Le but de cette technique est de retarder le plus possible les calculs. Cela signifie qu'on n'effectue les calculs que si on est certain qu'il peut exister un graphe-admissible meilleur que celui calculé. Une petite amélioration s'ajoute à cette approche et consiste à considérer en priorité parmi les sous-ensembles qui ont la même liste potentielle, ceux qui ont la liste réelle la plus élevée. Cette approche garde la complétude et permet de trouver un (ou plusieurs[3]) des graphes qui sont des graphes-solutions *GLM*. Cette heuristique est très bénéfique dans le cas où les contraintes du système hiérarchique sont assez consistantes entre elles.

L'application de cette amélioration sur l'exemple précédent est la suivante :

$$G = \left\{ \langle \{gaep_1^0, gaep_2^0\}, q_1 \rangle \right\},\ gaep_1^0 = (\{\{v_1 v_3\}, \{v_2\}, \{v_4\}), (requise, requise)),$$

$$gaep_2^0 = (\{\{v_2 v_3\}, \{\ \}, \{v_1 v_4\}), (requise, requise)),\ q_1 = \varnothing,$$

$$\text{liste-potentielle-étiquettes}\left(\langle \{gaep_1^0, gaep_2^0\}, q_1 \rangle\right) = \left(ep_1^0 \oplus \overline{q_1}\right) = (requise, requise).$$

Ici, G contient un seul sous-ensemble puisque tout couple dans ce sous-ensemble possède la même liste-potentielle-étiquettes. q_1 est la queue associée à ce sous-ensemble.

---

1. Ici un noeud de l'arbre est un état qui contient un ensemble de graphes-admissibles possédant la même liste d'étiquettes, contrairement à précédemment (dans la section 5.4.2) où un noeud est un état contenant un seul graphe-admissible.
2. Ici il s'agit d'ajouter d'une manière effective la contrainte sur chacun des couples dans ce sous-ensemble (en utilisant la conjonction et les différentes définitions de la consistance).
3. On peut avoir plusieurs *GLM* si le premier sous ensemble de G traité comporte plusieurs couples et qu'après l'ajout de la contrainte on obtient plusieurs couples ayant la liste d'étiquettes maximale.

On introduit dans la classe H_1 la contrainte c_{11} avec $M_{c_{11}} \equiv \{v_5 \rightarrow v_1\}$. G est mis à jour et son contenu est :

$$G = \left\{\langle\{gaep_1^1\}, q_1\rangle, \langle\{gaep_2^1\}, q_2\rangle\right\},\ q_1 = \varnothing,\ q_2 = \varnothing,$$

$$gaep_1^1 = ((\{v_5 v_3\}, \{v_2 v_1\}, \{v_4\}), (requise, requise, forte)),$$

$$gaep_2^1 = ((\{v_2 v_3\}, \{\ \}, \{v_1 v_4\}), (requise, requise)),$$

$$\text{liste-potentielle-étiquettes}\left(\langle\{gaep_1^1\}, q_1\rangle\right) = ep_1^1 \oplus \overline{q_1} = (requise, requise, forte),$$

$$\text{liste-potentielle-étiquettes}(\langle\{gaep_2^1\}, q_2\rangle) = ep_2^1 \oplus \overline{q_2} = (requise, requise).$$

$gaep_2^0 = gaep_2^1$ puisque la méthode de la contrainte c_{11} est non consistante avec une méthode requise du graphe-admissible dans le couple $gaep_2^0$. Ici $q_2 = \varnothing$ puisque cette queue a été associée au nouveau sous-ensemble contenant le couple $gaep_2^1$ après la tentative d'ajout de c_{11}.

On introduit maintenant dans la classe H_2 la contrainte c_{21} telle que: $M_{c_{21}} \equiv \{v_6 v_7 \rightarrow v_2\}$. G est mis à jour et son contenu est :

$$G = \left\{\langle\{gaep_1^2\}, q_1\rangle, \langle\{gaep_2^1\}, q_2\rangle\right\},\ q_1 = \varnothing,\ q_2 = \{c_{21}\},$$

$$gaep_1^2 = ((\{v_5 v_3\}, \{v_2 v_1\}, \{v_4\}), (requise, requise, forte)),$$

$$\text{liste-potentielle-étiquettes}(\langle\{gaep_1^2\}, q_1\rangle) = ep_1^2 \oplus \overline{q_1} = (requise, requise, forte),$$

$$\text{liste-potentielle-étiquettes}(\langle\{gaep_2^1\}, q_2\rangle) = ep_2^1 \oplus \overline{q_2} = (requise, requise, moyenne).$$

Il est à noter qu'ici il n'y a pas eu de création de nouveau couple à partir de $gaep_1^1$ et de la contrainte c_{21} puisque la méthode de cette contrainte est en conflit avec une des méthodes requises dans le couple.

Puisque la liste-potentielles-étiquettes (qui est aussi la liste-réelle-étiquettes) du couple $gaep_1^2$ est lexicographiquement supérieure à la liste-potentielle-étiquettes du sous-ensemble contenant le couple $gaep_2^1$, la procédure d'ajout s'arrête et donc un *GLM* est déjà dans le couple $gaep_1^2$.

On introduit maintenant dans la classe H_2 la contrainte c_{22} telle que: $M_{c_{22}} \equiv \{v_7 \rightarrow v_5\}$. G est mis à jour et son contenu est:

$$G = \left\{\langle\{gaep_1^3\}, q_1\rangle, \langle\{gaep_2^1\}, q_2\rangle\right\},\ q_1 = \varnothing,\ q_2 = \{c_{21}, c_{22}\},$$

$$gaep_1^3 = ((\{v_3 v_7\}, \{v_2 v_1 v_5\}, \{v_4\}), (requise, requise, forte, moyenne)),$$

$$\text{liste-potentielle-etiquettes}(\langle\{gaep_1^3\}, q_1\rangle) = ep_1^3 \oplus \overline{q_1} = (requise, requise, forte, moyenne),$$

$$\text{liste-potentielle-etiquettes}(\langle\{gaep_2^1\}, q_2\rangle) = ep_2^1 \oplus \overline{q_2} = (requise, requise, moyenne).$$

La procédure d'ajout s'arrête car il n'est pas possible de trouver un graphe-admissible meilleur que celui calculé dans $gaep_1^3$ puisque : $(ep_2^1 \oplus \overline{q_2}) \leq_{Lex} ep_1^3$ (*c.à.d* la liste-potentielle-étiquettes du deuxième sous-ensemble est non supérieure lexicographiquement à la liste-réelle-étiquettes du premier sous-ensemble dans *G)*.

Le retrait d'une contrainte c du système de contraintes hiérarchiques consiste à retirer toutes les méthodes de la contrainte c des différentes représentations des graphes-solutions dans G. Comme nous l'avons mentionné, l'ensemble G est partitionné en sous-ensembles, certains de ces sous-ensembles contiennent la contrainte c dans leur queue d'attente de contraintes à ajouter, et donc aucune des méthodes de la contrainte c ne figure dans les représentations des graphes-admissibles de ces sous-ensembles. Dans ce cas, la procédure de retrait de contrainte retire tout simplement la contrainte c de ces queues (puisqu'il n'y a pas eu un ajout effectif de c sur les différents graphes-admissibles de ces sous-ensembles).

L'ensemble G contient des sous-ensembles qui possèdent dans leurs différentes représentations une des méthodes de la contrainte c (c'est-à-dire qu'il y a eu un ajout effectif de la contrainte c sur les différents graphes-admissibles des couples de ces sous-ensembles[1]). Dans ce cas, la procédure de retrait procède de la manière suivante : pour un graphe-admissible qui inclut dans sa représentation une des méthodes de la contrainte c, ce graphe-admissible sera enlevé du sous-ensemble où il se trouve, la méthode de la contrainte c sera retirée de ce graphe-admissible en utilisant l'opérateur — et une nouvelle représentation sera déduite. Cette dernière est prise en compte dans ce sous-ensemble s'il n'existe aucune autre représentation dans les sous-ensembles de G qui la recouvre[2] (pour garder la maximalité des graphes-admissibles dans G). L'ensemble G après cette opération est ordonné selon la liste-potentielle-étiquettes des sous-ensembles qui le composent. Après cet ordonnancement, il se peut que la queue des contraintes à ajouter du premier sous-ensemble dans G soit non vide. Dans ce cas, l'opération d'ajout effectif des contraintes de cette queue sur les couples de ce sous-ensemble est déclenchée. La procédure s'achève lorsque la queue du premier sous-ensemble dans G est vide (c.à.d. liste-potentielle-étiquette = liste-réelle-étiquettes). La procédure retourne le premier sous-ensemble dans G. Celui ci contient les graphes-solutions *GLM* du système de contraintes hiérarchiques.

Exemple :

Hypothèses : $H = \{H_1, H_2, H_3\}$ avec H_1 , H_2 et H_3 sont les classes des contraintes flexibles. H_1 est la classe des contraintes étiquetées par l'étiquette *forte* et contenant les contraintes suivantes : c_{11}, c_{12}, c_{13}. H_2 est la classe des contraintes étiquetées par l'étiquette *moyenne* et contenant la contrainte c_{21}. H_3 est la classe des contraintes étiquetées par l'étiquette *faible* et contenant les contraintes suivantes : c_{31}, c_{32}, c_{33}. Chaque contrainte c_{ij} contient une seule méthode notée m_{ij}[3]. On a les relations suivantes : *Consistante*(m_{11}, m_{12}), *Consistante*$(m_{11} \wedge m_{12}, m_{13})$, *Consistante*$(m_{11} \wedge m_{12}, m_{21})$, *Consistante*$(m_{11} \wedge m_{13}, m_{21})$, *Consistante*$(m_{31} \wedge m_{32})$.

L'ensemble G après l'introduction successive des contraintes c_{11}, c_{12}, c_{13}, c_{21}, c_{31}, c_{32}, c_{33} est de la forme :

$$G = \left\{ \langle \{gaep_1^7\}, q_1 \rangle, \langle \{gaep_1^4, gaep_2^4\}, q_2 \rangle, \langle \{gaep_1^5\}, q_3 \rangle, \langle \{gaep_1^6\}, q_4 \rangle, \langle \{gaep_1^7\}, q_5 \rangle \right\},$$

$$gaep_1^7 = ((m_{11} \wedge m_{12} \wedge m_{13}), (\overline{c_{11}} \oplus \overline{c_{12}} \oplus \overline{c_{13}})) \qquad q_1 = \varnothing,$$

$$gaep_1^4 = ((m_{11} \wedge m_{12} \wedge m_{21}), (\overline{c_{11}} \oplus \overline{c_{12}} \oplus \overline{c_{21}})) \qquad q_2 = \{c_{31}, c_{32}, c_{33}\},$$

$$gaep_2^4 = ((m_{11} \wedge m_{13} \wedge m_{21}), (\overline{c_{11}} \oplus \overline{c_{13}} \oplus \overline{c_{21}})),$$

$$gaep_1^5 = ((m_{31}), (\overline{c_{31}})) \qquad q_3 = \{c_{32}, c_{33}\},$$

$$gaep_1^6 = ((m_{32}), (\overline{c_{32}})) \qquad q_4 = \{c_{33}\},$$

$$gaep_1^7 = ((m_{33}), (\overline{c_{33}})) \qquad q_5 = \varnothing.$$

1. Parmi ces sous-ensembles, certains peuvent parfaitement avoir des queues non vides.
2. Il est à noter ici que tous les sous-ensemble de G sont concernés
3. Ici on ne va donner le contenu des méthodes, mais on va supposer la consistance de quelques conjonctions entres ces méthodes.

On suppose maintenant que l'on va retirer la contrainte c_{21} du système. Les deux couples du deuxième sous-ensemble de G contiennent la méthode de cette contrainte. Ces deux couples sont retirés de ce sous-ensemble et les couples déduits après enlèvement de la méthode m_{21} sont respectivement : $((m_{11} \wedge m_{12}),(\overline{c_{11}} \oplus \overline{c_{12}}))$, $((m_{11} \wedge m_{13}),(\overline{c_{11}} \oplus \overline{c_{13}}))$. Ces deux couples déduits ne sont pas remis dans S puisque le couple dans le premier sous-ensemble de G les recouvre. L'ensemble G après cette opération est :

$$G = \left\{ \langle \{gaep_1^7\}, q_1 \rangle, \langle \{gaep_1^5\}, q_3 \rangle, \langle \{gaep_1^6\}, q_4 \rangle, \langle \{gaep_1^7\}, q_5 \rangle \right\},$$
$$gaep_1^7 = ((m_{11} \wedge m_{12} \wedge m_{13}),(\overline{c_{11}} \oplus \overline{c_{12}} \oplus \overline{c_{13}})) \qquad q_1 = \varnothing,$$
$$gaep_1^5 = ((m_{31}),(\overline{c_{31}})) \qquad q_3 = \{c_{32}, c_{33}\},$$
$$gaep_1^6 = ((m_{32}),(\overline{c_{32}})) \qquad q_4 = \{c_{33}\},$$
$$gaep_1^7 = ((m_{33}),(\overline{c_{33}})) \qquad q_5 = \varnothing.$$

On suppose maintenant que l'on doit retirer la contrainte c_{33}. Les queues q_3 et q_4 ainsi que le dernier sous-ensemble de G sont concernés par ce retrait. Le résultat est :

$$G = \left\{ \langle \{gaep_1^7\}, q_1 \rangle, \langle \{gaep_1^5\}, q_3 \rangle, \langle \{gaep_1^6\}, q_4 \rangle \right\}$$
$$gaep_1^7 = ((m_{11} \wedge m_{12} \wedge m_{13}),(\overline{c_{11}} \oplus \overline{c_{12}} \oplus \overline{c_{13}})) \qquad q_1 = \varnothing$$
$$gaep_1^5 = ((m_{31}),(\overline{c_{31}})) \qquad q_3 = \{c_{32}\}$$
$$gaep_1^6 = ((m_{32}),(\overline{c_{32}})) \qquad q_4 = \varnothing.$$

Et enfin, on suppose que l'on doit retirer successivement les trois contraintes de la classe H_1. Le résultat après ce retrait est :

$$G = \{\langle \{gaep_1^6\}, q_3 \rangle\}$$
$$gaep_1^6 = ((m_{31} \wedge m_{32}),(\overline{c_{31}} \oplus \overline{c_{32}})) \qquad q_3 = \varnothing.$$

Le retrait de ces trois contraintes successives de G a produit l'enlèvement complet du premier sous-ensemble de G. Il restait donc dans G en première position le sous-ensemble ayant pour queue q_3 et en deuxième position le sous-ensemble ayant pour queue q_4 (puisque l'ensemble G doit être à tout moment ordonné selon la liste-potentielle-étiquettes des sous-ensembles qui le composent). La procédure d'ajout effectif a été déclenchée puisque la queue q_3 était non vide. Il se trouve que par hypothèse on a la méthode m_{31} consistante avec la méthode m_{32}. Par conséquent, le couple comportant la méthode m_{31} est mis-à-jour et comporte maintenant la conjonction de ces deux méthodes. Ce couple mis-à-jour recouvre un couple existant dans G, c'est ainsi que le couple couvert est retiré de G.

Synthèse du chapitre

L'objet de ce cinquième chapitre a été le développement du noyau d'un nouveau résolveur incrémental pour la résolution d'une hiérarchie de contraintes fonctionnelles. L'aproche algorithmique utilisée ici est celle de l'algorithme du type "meilleur d'abord" où un noeud représentant un état qui est un graphe de méthodes ou un ensemble de graphes de méthodes, une branche représente un ajout ou un retrait de contrainte et enfin la fonction d'évaluation qui dépend du critère global *Nombre-Contraintes-Non-Satisfaites*. Ce résolveur implémente ce critère global. Nous avons vu que ce critère de comparaison permet de mieux discriminer l'ensemble de solutions qui satisfont les contraintes du niveau H_0 qu'un critère local (voir l'exemple de la section 5.2.1) et permet ainsi de donner des solutions de bonne qualité lorsqu'il s'agit d'un système sur-contraint (c.à.d. les solutions trouvées sont meilleures en terme du nombre de contraintes satisfaites en respectant la hiérarchie).

Les résolveurs qui implémentent des critères locaux ne garantissent pas des solutions de meilleure qualité, puisqu'à l'introduction d'une contrainte au niveau H_i de la hiérarchie, si cette contrainte est en conflit avec une autre contrainte (du niveau H_j, telle que: $j=i$ ou $j<i$) active dans le graphe admissible alors elle sera rejetée sans se préoccuper si elle pourrait dans l'avenir construire (après l'ajout ou le retrait d'autres contraintes) une solution meilleure que la solution courante.

Pour construire un graphe-solution GLM, ce résolveur utilise les différents objets suivants :

- la définition d'un *GLM* décrite dans la section 5.2 à savoir :

  pas de conflit entre les méthodes,

  pas de circuit de méthodes,

  le nombre de contraintes actives par niveau est maximal.

- les outils définis dans la section 5.3 à savoir :

  la représentation d'une méthode et d'un ensemble de méthodes (efficacité de traitement),

  les composantes connexes (efficacité de traitement),

  les opérateurs $\wedge$, $\oplus$ et — (opérations sur les graphes-admissibles),

  le prédicat *Pas-De-Conflit* (détection de conflit),

  le prédicat *Pas-De-Circuit-bis* (détection de circuit),

  Consistant,

  les deux théorèmes 5.2 et 5.3 (efficacité de traitement).

- une technique d'optimisation qui consiste à ne créer un état que s'il est maximal (contribue à réduire la complexité en espace mémoire)

- la fonction d'évaluation $f=g+h$ (permet de déterminer le meilleur noeud à développer dans l'arbre de recherche) :

  g est la liste ordonnée des étiquettes des méthodes actives dans un état

  h est la liste ordonnée des étiquettes des contraintes non actives (mises dans la queue)

Houria est un résolveur incrémental qui maintient les contraintes d'une hiérarchie dans un ensemble G de graphes-admissibles en construisant un (ou plusieurs) graphe-solution(s) *GLM*. Une fois le graphe-solution *GLM* construit, Houria exécute ses méthodes actives pour satisfaire les contraintes actives (nous rappelons qu'une contrainte est active dans un graphe-admissible si elle possède une méthode active dans ce graphe).

On suppose qu'initialement, l'ensemble G des graphes-admissibles contient l'ensemble des graphes-admissibles qui correspondent aux contraintes requises de la classe H_0 (cette supposition est réfutée dans les procédures *Ajout-contrainte* et *Retrait-contrainte* que nous donnerons au chapitre 6, elle est mise ici uniquement dans un but pédagogique). Houria est appelé par deux procédures, *Ajout-Contrainte* et *Retrait-Contrainte*, sur chaque graphe-admissible dans G. C'est ainsi qu'il met à jour ces graphes-admissibles dans G en tenant compte de leurs étiquettes et de l'étiquette de la contrainte à ajouter ou à retirer pour déterminer le graphe-solution *GLM*.

Si la hiérarchie de contraintes contient une solution acyclique alors on garantit que ce résolveur la trouve, tandis que si la hiérarchie de contrainte contient uniquement des solutions cycliques, et du fait qu'on casse les circuits dans la construction du *GLM*, on ne trouve que des solutions proches de ces solutions cycliques (proche en terme du nombre de contraintes). On peut trouver ces solutions cycliques s'il l'on conçoit un sous-résolveur pour la résolution des circuits détectés par le prédicat *Pas-De-Circuit-Bis*.

Nous avons défini d'une manière générale (exemples : *Pas-De-Circuit*, —) et ensuite spécifique (exemples : *Pas-De-Circuit-Bis*, —) un ensemble d'outils qui permet la construction (ou la planification) d'un graphe dit graphe lexicographiquement meilleur. Ce dernier permet de déterminer une solution (ou valuation) qui satisfait les contraintes de la hiérarchie. Quelques techniques d'optimisation sont intégrées dans cet algorithme, ceci pour trouver une solution le plus rapidement possible tout en occupant un espace mémoire réduit pour représenter les graphes-admissibles.

Le mode de représentation des contraintes utilisé par cet algorithme est souple : il permet de manipuler les graphes-admissible comme des ensembles de variables, ce qui rend les calculs moins coûteux en temps.

Ce résolveur qui implémente un critère global manipule plusieurs graphes-admissibles dans un système hiérarchique, ceci est particulièrement intéressant lorsque l'utilisateur désire avoir plus qu'une solution et explorer les différentes solutions alternatives.

L'éventail des idées et méthodes proposées dans cet algorithme permet la résolution correcte d'une hiérarchie de contraintes fonctionnelles. Le noyau de cet algorithme tel qu'il est conçu (à une petite extension près que nous allons présenter au chapitre suivant) permet l'intégration d'autres critères globaux de comparaison que nous allons définir dans le chapitre suivant.

6 Généralisation de Houria pour l'intégration d'autres critères de comparaison.

Un des points importants de la théorie des hiérarchies de contraintes est le fait de pouvoir définir différents types de comparateurs. Ces différents types de comparateurs doivent être jugés d'après la réponse à la question suivante : à quel degré représentent-ils l'intention de l'utilisateur dans des problèmes réels ? Le chapitre précédent décrit notre résolveur Houria basé sur le critère global *Nombre-Contraintes-Non-Satisfaites*. Ce résolveur permet de trouver des solutions qui sont plus intuitives que celles obtenues par un résolveur intégrant un comparateur local (Pour un même système surcontraint, les solutions trouvées par le comparateur de ce résolveur satisfont plus de contraintes que celles trouvées par un critère local). Cependant, ce résolveur de même que les autres résolveurs existants, supposent que les contraintes d'une hiérarchie donnée sont uniquement étiquetées et non pondérées par des poids réels.

L'objectif de ce chapitre est de surmonter cette restriction et donc de pouvoir résoudre des hiérarchies où les contraintes sont étiquetées et pondérées par des poids (ou associées à des indices) réels différents. Plusieurs applications illustrent ce besoin, on citera ici deux exemples. Le premier type d'application est celui où l'utilisateur a besoin d'exprimer des contraintes dites de remplacement. Au sein d'une même classe dans la hiérarchie, on peut avoir deux types de contraintes : celles associées à un indice[1] fort dont la satisfaction sera considérée prioritaire, et celles associées à des indices moins forts et qui sont considérées comme des contraintes de remplacement. Dans ce cas, le résolveur doit d'abord considérer la satisfaction des contraintes associées à un indice fort et dans les cas d'échec (si cette satisfaction ne peut pas avoir lieu), considérer la satisfaction des contraintes de remplacement. Le deuxième type d'application est celui où le poids d'une contrainte traduit son degré d'importance dans la classe de la hiérarchie où elle se trouve. Dans ce cas et selon le comparateur utilisé, les contraintes pondérées par des poids forts ne sont pas forcément à satisfaire en priorité (si l'on suppose qu'on utilise le comparateur *Somme-Pondérée* en utilisant l'erreur prédicat, et qu'on a la disjonction exclusive entre la satisfaction d'une contrainte pondérée à 0.9 et de deux contraintes pondérées à 0.5 chacune, alors ces dernières sont prioritaires pour la satisfaction).

1. Ici on préfère utiliser le terme indice au terme poids.

Ce chapitre est composé de cinq parties : la première et la deuxième parties décrivent les définitions d'autres critères de comparaison globaux intégrés dans Houria ainsi que les relations entre les ensembles de solutions produits par ces différents critères et celui produit par le premier critère décrit au chapitre précédent. Chacune de ces deux parties décrit également les techniques utilisées pour la construction des graphes solutions correspondant au critère intégré. La troisième partie décrit les procédures généralisées (paramétrées par un type de comparateur) d'ajout et de retrait d'une contrainte à la hiérarchie. La quatrième partie traite la complexité et l'implémentation de Houria ainsi des mesures effectuées sur des problèmes générés aléatoirement. Enfin, la cinquième partie décrit une comparaison fonctionnelle entre Houria et les autres algorithmes de propagation locale présentés auparavant.

6.1 Le deuxième critère de comparaison intégré dans Houria

6.1.1 Définition formelle

Le deuxième critère de comparaison dans cet algorithme est le comparateur *Cardinal-Pire-Cas*. Ce comparateur utilise le nombre de contraintes satisfaites associées au plus fort indice de chaque classe dans la hiérarchie. L'ensemble des solutions obtenues par l'utilisation de ce comparateur est généralement de taille plus petite que celle de l'ensemble obtenu en utilisant le critère *Localement-Prédicat-Meilleur* ou encore celle de l'ensemble obtenu en utilisant le critère global *Pire-Cas*. Avant de définir ce critère, on présentera d'abord la définition du comparateur *Pire-Cas*.

Définition 6.1:

Une valuation θ est meilleure selon le critère *Pire-Cas* qu'une autre valuation η, si et seulement si, pour chacun des niveaux jusqu'au niveau *k-1*, le plus fort indice parmi ceux des contraintes non satisfaites après application de θ est égal à celui après application de η, et au niveau *k*, cet indice est strictement inférieur.

$$Pire\text{-}Cas(\theta, \eta, H) \Leftrightarrow Globalement\text{-}Meilleure(\eta, \theta, H, g), \text{ avec } g(\phi, H_i) \equiv Max\ \{c\ e(c\ \phi) / (c \in H_i)\}$$

avec e(c φ) est l'erreur prédicat (qui rend 0 si la contrainte est satisfaite et 1 sinon).

Définition 6.2:

Une valuation θ est meilleure selon le critère *Cardinal-Pire-Cas* qu'une autre valuation η, si et seulement si, pour chacun des niveaux jusqu'au niveau *k-1*, le nombre de contraintes non satisfaites associées au plus fort indice après application de θ est égal à celui après application de η, et au niveau *k*, ce nombre est strictement inférieur.

$$Cardinal\text{-}Pire\text{-}Cas(\theta, \eta, H) \Leftrightarrow Globalement\text{-}Meilleure(\eta, \theta, H, g), \text{ avec}$$

$$g(\phi, H_i) \equiv (Max\ \{\ \overline{c}\ e(c\ \phi) / (c \in H_i)\}, Card(Max^1\ \{\ \overline{c}\ e(c\ \phi) / (c \in H_i)\})).$$

Comme il est mentionné au début de ce chapitre, ce comparateur peut être utilisé dans des application où l'utilisateur a besoin d'exprimer des contraintes dites de remplacement. Par exemple, si l'on considère la hiérarchie suivante :

$H = \{ H_1 \}$ où H_1 contient les quatre contraintes suivantes : *{x.A=x.souris}, {y.A = y.souris}, {x.B=x.souris}, {y.B=y.souris}* qui ont respectivement pour indices *0.9, 0.9, 0.8, 0.8*.

Ici, l'utilisation de ce comparateur voudrait dire :

1. Par abus de langage, dans cette partie de la définition on supose que la fonction *Max* rend l'ensemble des maximaux et non un seul

on préfère une valuation qui satisfait les deux contraintes *{x.A=x.souris}* et *{y.A = y.souris}* à une valuation qui satisfait uniquement une de ces deux contraintes (puisqu'on souhaite déplacer par la souris une ligne ayant pour extrémités les points A et B par le point A) ou encore une valuation qui satisfait les contraintes *{x.B=x.souris}, {y.B=y.souris},*

aussi, dans le cas ou il est impossible de satisfaire les contraintes *{x.A=x.souris}* et *{y.A = y.souris}* alors on va essayer de satisfaire les autres contraintes de remplacement.

Le comparateur *Cardinal-Pire-Cas* est un comparateur global qui discrimine mieux l'ensemble des valuations qui satisfont les contraintes requises que le comparateur *Pire-Cas* (ou que le comparateur *Localement-Prédicat-Meilleur*). Par conséquent, la taille de l'ensemble des solutions d'une hiérarchie en utilisant ce critère est généralement plus petite que si l'on utilise le critère *Pire-Cas*. Pour illustrer ceci considérons l'exemple suivant :

Exemple :

Soient $H = \{H_0, H_1, H_2\}$ avec H_0 la classe des contraintes requises. H_1 est la classe des contraintes étiquetées par l'étiquette *forte*. H_2 est la classe des contraintes étiquetées par l'étiquette *moyenne*. H_0 contient deux contraintes c_{01} et c_{02}. H_1 contient quatre contraintes $c_{11}, c_{12}, c_{13}, c_{14}$ et ont respectivement pour indices *0.9, 0.9, 0.9, 0.8*. H_2 contient trois contraintes c_{21}, c_{22}, c_{23} qui ont respectivement pour indices *0.9, 0.9, 0.9*. On rappele que l'erreur prédicat utilisée rend 0 si la contrainte est satisfaite et 1 sinon. On suppose aussi que :

Les valuations θ, φ et τ ne satisfont pas respectivement les ensembles de contraintes : $\{c_{01}, c_{02}, c_{11}, c_{12}, c_{14}, c_{21}, c_{22}\}$, $\{c_{01}, c_{02}, c_{11}, c_{14}, c_{21}, c_{22}\}$ et $\{c_{01}, c_{02}, c_{11}, c_{13}, c_{14}, c_{21}\}$.

Les trois valuations satisfont les deux contraintes requises dans H_0 et donc $S_0 = \{\theta, \phi, \tau\}$. Pour calculer S, les séquences de satisfaction par niveau de chaque valuation en utilisant *Pire-Cas* sont :

$g(\theta, H_1) = Max(1 * 0.9, 1 * 0.9, 0 * 0.9, 1 * 0.8) = 0.9$

$g(\theta, H_2) = Max(1 * 0.9, 1 * 0.9, 0 * 0.9) = 0.9$

$g(\phi, H_1) = Max(1 * 0.9, 0 * 0.9, 0 * 0.9, 1 * 0.8) = 0.9$

$g(\phi, H_2) = Max(1 * 0.9, 1 * 0.9, 0 * 0.9) = 0.9$

$g(\tau, H_1) = Max(1 * 0.9, 0 * 0.9, 1 * 0.9, 1 * 0.8) = 0.9$

$g(\tau, H_2) = Max(1 * 0.9, 0 * 0.9, 0 * 0.9) = 0.9$

On observe que : $g(\theta, H_1) = g(\phi, H_1) = g(\tau, H_1)$ et $g(\theta, H_2) = g(\phi, H_2) = g(\tau, H_2)$. Par conséquent $S = \{\theta, \phi, \tau\}$.

Les séquences de satisfaction par niveau de chaque valuation en utilisant *Cardinal-Pire-Cas* sont :

$g(\theta, H_1) = (Max(1*0.9, 1*0.9, 0*0.9, 1*0.8), Card(Max(1*0.9, 1*0.9, 0*0.9, 1*0.8))) = (0.9, 2)$

$g(\theta, H_2) = (Max(1 * 0.9, 1 * 0.9, 0 * 0.9), Card(Max(1 * 0.9, 1 * 0.9, 0 * 0.9))) = (0.9, 2)$

$g(\phi, H_1) = (Max(1 * 0.9, 0 * 0.9, 0 * 0.9, 1 * 0.8), Card(Max(1 * 0.9, 0 * 0.9, 0 * 0.9, 1 * 0.8))) = (0.9, 1)$

$g(\phi, H_2) = (Max(1 * 0.9, 1 * 0.9, 0 * 0.9), Card(Max(1 * 0.9, 1 * 0.9, 0 * 0.9))) = (0.9, 2)$

$g(\tau, H_1) = (Max(1*0.9, 0*0.9, 1*0.9, 1*0.8), Card(Max(1*0.9, 0*0.9, 1*0.9, 1*0.8))) = (0.9, 2)$

$g(\tau, H_2) = (Max(1 * 0.9, 0 * 0.9, 0 * 0.9), Card(Max(1 * 0.9, 0 * 0.9, 0 * 0.9))) = (0.9, 1)$

On observe que : $g(\phi, H_1) <_{lex} g(\theta, H_1)$, $g(\tau, H_1) = g(\theta, H_1)$ et $g(\tau, H_2) <_{lex} g(\theta, H_2)$. Par conséquent $S=\{\phi\}$.

Proposition 6.1:

$\forall\theta, \eta\ \forall H$ *Pire-Cas*$(\theta, \eta, H) \rightarrow$ *Cardinal-Pire-Cas*$(\theta, \eta, H) \vee^1$ *Cardinal-Pire-Cas*(η, θ, H).

Preuve :

Supposons que *Pire-Cas*(θ, η, H) est vrai, et donc qu'il existe un niveau $k>0$ dans H tel qu'après application de chacune des contraintes sur θ jusqu'au niveau $k-1$, le plus fort indice parmi ceux des contraintes non satisfaites est égal à celui obtenu après application sur η. Ceci veut dire que l'on est dans un des deux cas suivants :

1. Après application de θ, le nombre de contraintes non satisfaites ayant le plus fort indice par niveau jusqu'au niveau $k-1$ est égal à celui obtenu après application de η. Dans ce cas au niveau k, après application de θ, il existe au moins une contrainte satisfaite par θ (et non par η) telle qu'elle possède un indice plus fort que celui de toute contrainte satisfaite par η (puisque *Pire-Cas*(θ, η, H)). Par conséquent, on a bien *Cardinal-Pire-Cas*(θ, η, H) et donc l'implication est vraie.
2. Après application de θ, il existe un niveau compris entre 1 et $k-1$ pour lequel le nombre de contraintes non satisfaites ayant le plus fort indice est soit supérieur soit inférieur (mais non égal) à celui obtenu après application de η. Si ce nombre est supérieur, alors *Cardinal-Pire-Cas*(η, θ, H) est vrai. Si ce nombre est inférieur alors *Cardinal-Pire-Cas*(θ, η, H) est vrai. Par conséquent, dans les deux cas l'implication est vraie[2].

Après avoir défini ce critère global de comparaison, la question que l'on peut se poser est : *Peut-on avoir le même ensemble de solutions que celui produit par le comparateur Cardinal-Pire-Cas en subdivisant la hiérarchie et en utilisant le comparateur Nombre-Contraintes-Non-Satisfaites* ? La réponse à cette question est négative comme le montre le paragraphe suivant :

Ici, on donnera un exemple pour lequel l'ensemble de solutions obtenu dans une hiérarchie en utilisant le comparateur *Cardinal-Pire-Cas* est différent de celui obtenu en subdivisant la même hiérarchie et en utilisant le comparateur *Nombre-Contraintes-Non-Satisfaites*.

Soient $H= \{H_0, H_1, H_2, H_3\}$ avec H_0 H_1, H_2, H_3 des classes contenant respectivement des contraintes étiquetées par les étiquettes *requise*, *forte*, *moyenne* et *faible*. H_0 contient deux contraintes c_{01} et c_{02}. H_1 contient deux contraintes c_{11} et c_{12} et sont associées respectivement aux indices *0.9*, *0.9*. H_2 contient trois contraintes c_{21}, c_{22} et c_{23} et sont associées respectivement aux indices *0.9*, *0.9*, *0.8*. H_3 contient deux contraintes c_{31} et c_{32} et sont associées respectivement aux indices *0.9*, *0.8*. On suppose aussi que :

Les valuations θ et τ satisfont respectivement les ensembles de contraintes : $\{c_{01}, c_{02}, c_{11}, c_{21}, c_{31}\}$, $\{c_{01}, c_{02}, c_{12}, c_{22}, c_{23}, c_{32}\}$.

Les deux valuations satisfont les deux contraintes requises dans H_0 et donc $S_0 = \{\theta, \tau\}$. les séquences de satisfaction par niveau de ces deux valuations en utilisant le comparateur *Cardinal-Pire-Cas* sont :

1. Ici il s'agit d'un ou exclusif.
2. C'est ce deuxième cas qui permet de mieux discriminer l'ensemble des solutions dans S_0.

$g(\theta, H_1) = (Max(0 * 0.9, 1 * 0.9), Card(Max(0 * 0.9, 1 * 0.9))) = (0.9, 1)$

$g(\theta, H_2) = (Max(0 * 0.9, 1 * 0.9, 1 * 0.8), Card(Max(0 * 0.9, 1 * 0.9, 1 * 0.8))) = (0.9, 1)$

$g(\theta, H_3) = (Max(0 * 0.9, 1 * 0.8), Card(Max(0 * 0.9, 1 * 0.8))) = (0.8, 1)$

$g(\tau, H_1) = (Max(1 * 0.9, 0 * 0.9), Card(Max(1 * 0.9, 0 * 0.9))) = (0.9, 1)$

$g(\tau, H_2) = (Max(1 * 0.9, 0 * 0.9, 0 * 0.8), Card(Max(1 * 0.9, 0 * 0.9, 0 * 0.8))) = (0.9, 1)$

$g(\tau, H_3) = (Max(1 * 0.9, 0 * 0.8), Card(Max(1 * 0.9, 0 * 0.8))) = (0.9, 1)$

On observe que : $g(\tau, H_1) = g(\theta, H_1)$, $g(\tau, H_2) = g(\theta, H_2)$ et $g(\theta, H_3) <_{lex} g(\tau, H_3)$. Par conséquent la valuation θ est préférée à la valuation τ et donc $S=\{\theta\}$ (Pour cet exemple le critère *Pire-Cas* produit le même ensemble de solution que le critère *Cardinal-Pire-Cas*).

Après subdivision de cette hiérarchie on obtient: $H=\{H_0, H_{1\text{-}0.9}, H_{2\text{-}0.9}, H_{2\text{-}0.8}, H_{3\text{-}0.9}, H_{3\text{-}0.8}\}$. $H_{1\text{-}0.9}$ contient les deux contraintes c_{11} et c_{12}. $H_{2\text{-}0.9}$ contient les deux contraintes c_{21} et c_{22}. $H_{2\text{-}0.8}$ contient la contrainte c_{23}. $H_{3\text{-}0.9}$ contient la contrainte c_{31}. $H_{3\text{-}0.8}$ contient la contrainte c_{32}. Par utilisation du critère *Nombre-Contraintes-Non-Satisfaites*, les séquences d'erreurs obtenues à la suite de l'application respectivement des valuations θ et τ sur cette hiérarchie subdivisée sont : (0, 1, 1, 1, 0, 1) et (0, 1, 1, 0, 1, 0). Le deuxième tuple étant lexicographiquement inférieur au premier, par conséquent on déduit que la valuation τ est préférée à la valuation θ et donc $S=\{\tau\}$.

La section suivante décrit la technique utilisée pour planifier les graphes solutions correspondant à ce deuxième critère intégré.

6.1.2 Graphe solution correspondant au deuxième critère

En considérant ce nouveau critère de comparaison, la définition d'un graphe lexicographiquement meilleur (*GLM*) devient la suivante :

Définition 6.3 :

Soient deux graphes-admissibles s_1 et s_2. s_1 est lexicographiquement meilleur (*GLM*) que s_2 si et seulement si, pour tout niveau jusqu'au niveau k-1, le nombre de contraintes non actives associées au plus fort indice par niveau dans s_1 est égal au nombre de contraintes non actives dans s_2 associées au même indice, et au niveau k ce nombre est strictement inférieur.

Houria utilise les étiquettes et les indices associés aux contraintes pour planifier des graphes solutions. On rappelle ici la définition d'un graphe-solution : un graphe-admissible est un graphe-solution si et seulement s'il ne possède pas de cycle ni de conflit de méthodes, et de plus, il ne doit pas posséder une méthode inactive à un niveau k, telle que si on active cette méthode, elle génère un graphe lexicographiquement meilleur.

Etant donnée une hiérarchie de contraintes, pour obtenir un graphe solution correspondant au critère défini dans cette section, Houria effectue les deux étapes suivantes :

- Au sein d'une même classe, les contraintes attachées à des indices faibles sont laissées non satisfaites (c.à.d. non actives dans le graphe solution) dans le but de satisfaire (c.à.d. activer dans le graphe solution) celles attachées à des indices plus forts de la même classe ou d'autres classes.
- Entre deux classes, les contraintes faiblement étiquetées et rattachées à des indices faibles sont laissées non satisfaites pour satisfaire celles étiquetées fortement.

Par exemple[1], considérons le graphe admissible de la figure 23. Ce graphe n'est pas un graphe solution, car les méthodes des contraintes c_2 et c_4 sont en conflit sur la variable v_2. Puisque l'indice de la contrainte c_2 n'est pas le plus fort indice de la classe des contraintes étiquetées par l'étiquette *forte*, cette contrainte peut être désactivée (ceci correspond à la réalisation de la première étape décrite précédemment). Ceci produit le graphe-admissible de la figure 24 qui n'est pas un graphe solution car la contrainte c_3 peut être activée en désactivant la contrainte c_5, cette dernière n'étant pas rattachée au plus fort indice de la classe contenant les contraintes étiquetées par l'étiquette *moyenne* (ceci correspond à la réalisation de la deuxième étape décrite précédemment). En activant la contrainte c_3 et en désactivant la contrainte c_5, on aboutit au graphe solution de la figure 25. Ce dernier est un graphe solution puisqu'il n'existe aucune contrainte désactivée qui puisse être activée et produire un graphe meilleur que celui de la figure 25.

FIGURE 23 : Graphe-admissible non GLM

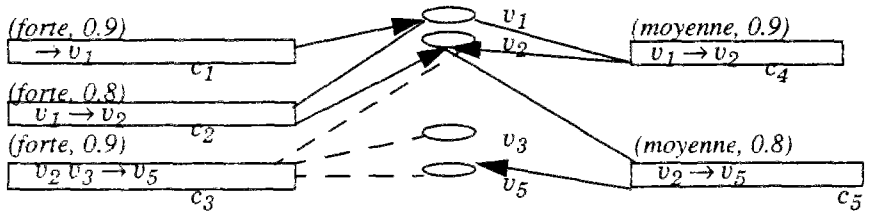


FIGURE 24 : Graphe-admissible non GLM

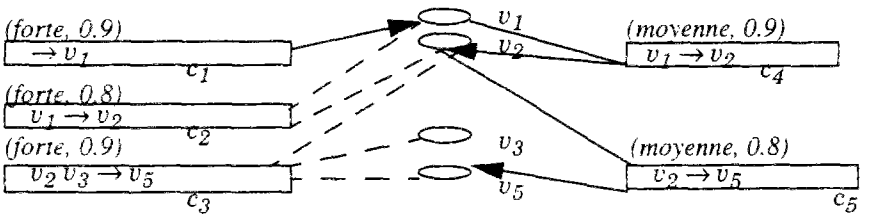
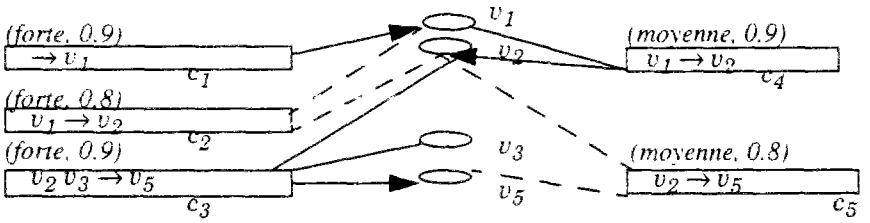


FIGURE 25 : Graphe solution



6.2 Le troisième critère de comparaison intégré dans Houria

6.2.1 Définition formelle

Le troisième critère de comparaison utilisé dans cet algorithme est le comparateur *Somme-Pondérée-Prédicat*. Ce comparateur utilise la somme des poids des contraintes satisfaites par niveau de la hiérarchie. L'ensemble des solutions obtenu par l'utilisation de ce comparateur est généralement

1. Pour simplifier l'exemple, on suppose que chaque contrainte contient une seule méthode.

de taille plus petite que celle de l'ensemble obtenu en utilisant le critère *Localement-Prédicat-Meilleur*. Ce critère est défini comme suit :

Définition 6.4:

Une valuation θ est considérée meilleure qu'une autre valuation η par le comparateur *Somme-Pondérée-Prédicat*, si et seulement si, pour chacun des niveaux jusqu'au niveau *k-1*, la somme des poids des contraintes non satisfaites après application de θ est égale à celui après application de η, et au niveau *k*, cet somme est strictement inférieure.

Somme-Pondérée-Prédicat(θ, η, *H*)⇔ *Globalement-Meilleur*(θ, η, *H*, *g*)

$$avec \quad g(\rho, C_i) \equiv \left(\sum_{c \in H_i} \bar{c} \, e \, (c\rho) \right).$$

On rappelle qu'avec cette définition, *S* ne doit pas contenir une solution moins bonne qu'une autre solution. *S* peut contenir plusieurs solutions (d'où l'existence de plusieurs graphes solutions GLM), aucune de ces solutions n'étant meilleure qu'une autre.

Proposition 6.2:

∀θ, η ∀*H Localement-Prédicat-Meilleur*(θ, η, *H*) → *Somme-Pondérée-Prédicat*(θ, η, *H*).

Preuve :

Supposons que *Localement-Prédicat-Meilleur*(θ, η, *H*) est vrai, et donc qu'il existe un niveau *k>0* dans *H* tel que le vecteur d'erreurs prédicats[1] obtenu après application des contraintes sur θ jusqu'au niveau *k-1* est égal à celui obtenu après application sur η. Ceci implique que : après application sur θ, la somme des éléments de ce vecteur pondérés par les poids correspondant aux contraintes jusqu'au niveau *k-1* est la même que celle obtenue par application de η. De plus au niveau *k*, et par utilisation de *Localement-Prédicat-Meilleur*(θ, η, *H*), toute contrainte satisfaite par η est satisfaite par θ. En plus, θ satisfait au moins une contrainte de plus que η. Donc, le vecteur d'erreurs prédicats obtenu après application sur θ est strictement inférieur lexicographiquement à celui obtenu après application de η. Ceci implique aussi que la somme des éléments de ce vecteur pondérée par les poids correspondant aux contraintes du niveau *k* est strictement inférieure à celle obtenue par application de η. Par conséquent, on a bien *Somme-Pondérée-Prédicat*(θ, η, *H*).

Corollaire 6.2:

Pour une hiérarchie de contraintes donnée, soit S_{LPM} l'ensemble des solutions obtenu par l'utilisation du comparateur *localement-meilleur*, et S_{SPP} celui obtenu par l'utilisation du comparateur *somme-pondérée*, on a : $S_{SPP} \subseteq S_{LPM}$.

Le comparateur *Nombre-Contraintes-Non-Satisfaites* est un cas particulier du comparateur *Somme-Pondérée-Prédicat* utilisant le même poids pour chacune des contrainte de chaque niveau de la hiérarchie. Dans ces conditions $S_{NCNS} \equiv S_{SPP}$.

La section suivante décrit la technique utilisée pour planifier les graphes solutions correspondant à ce troisième critère.

1. Chaque élément dans ce vecteur est égal à 0 si la contrainte est satisfaite et à 1 sinon.

6.2.2 Graphe solution correspondant au troisième critère

En considérant ce critère de comparaison, la définition d'un graphe lexicographiquement meilleur (*GLM*) devient la suivante :

Définition 6.4 :

Soient deux graphes-admissibles s_1 et s_2, s_1 est lexicographiquement meilleur (*GLM*) que s_2 si et seulement si, pour tout niveau jusqu'au niveau *k-1*, la somme des poids des contraintes par niveau non-actives dans s_1 est égale à celle des contraintes non-actives par niveau dans s_2, et au niveau k cette somme est strictement inférieure.

Etant donnée une hiérarchie de contraintes, pour obtenir un graphe solution correspondant au critère défini dans cette section, Houria effectue les deux étapes suivantes :

- Au sein d'une même classe, les contraintes pondérées par des poids faibles sont laissées non satisfaites (c.à.d. non actives dans le graphe solution) dans le but de satisfaire (c.à.d. activer dans le graphe solution) celles pondérées par des poids plus forts.
- Entre deux classes, les contraintes faiblement étiquetées sont laissées non satisfaites pour satisfaire celles étiquetées fortement.

Par exemple, considérons le graphe-admissible de la figure 26. Ce graphe-admissible n'est pas un graphe-solution puisque les méthodes des contraintes c_1 et c_2 sont sur conflit en la variable v_3. Puisque le poids de la contrainte c_2 est plus petit que celui de la contrainte c_1, la contrainte c_2 est désactivée (ceci correspond à la réalisation de la première étape décrite précédemment). Ceci produit le graphe-admissible de la figure 27.

Le graphe-admissible de la figure 27 n'est pas un graphe-solution puisque la contrainte c_3 peut être activée en désactivant la contrainte c_4 du fait que cette dernière est étiquetée par une étiquette plus faible que celle de c_3 (ceci correspond à la réalisation de la deuxième étape décrite précédemment).

En activant les contraintes c_3 et c_5 et en désactivant la contrainte c_4, le graphe-admissible obtenu est celui de la figure 28. Ce dernier est un graphe-solution puisqu'il n'existe aucune contrainte désactivée qui peut être activée et produire un graphe meilleur que celui-là.

FIGURE 26 : Graphe admissible non GLM

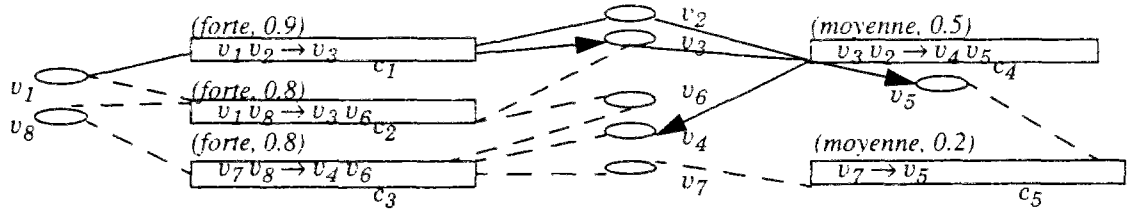


FIGURE 27 : Graphe-admissible non GLM

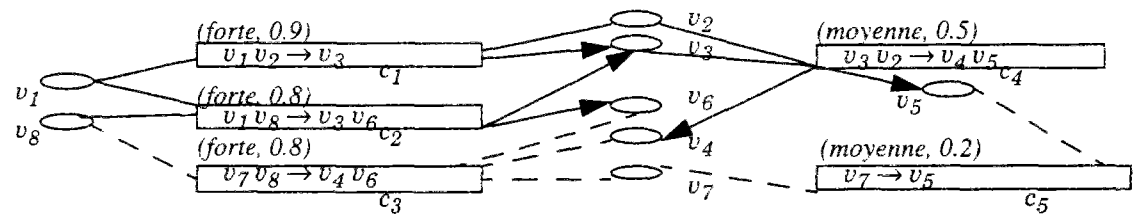
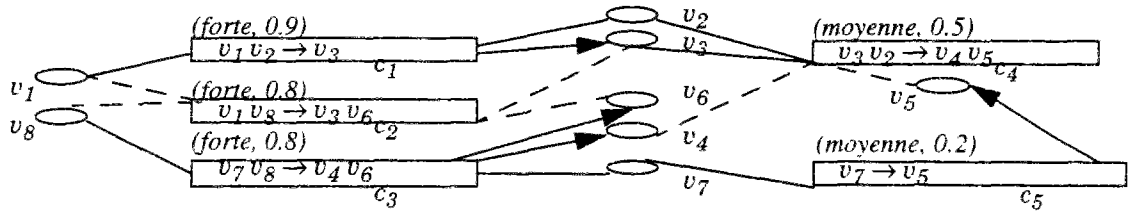


FIGURE 28 : Graphe- solution



6.3 Procédures généralisées d'ajout et du retrait de contrainte

Houria est appelé par deux procédures, *Ajout-Contrainte* et *Retrait-Contrainte*, sur chaque graphe-admissible dans G. C'est ainsi qu'il met à jour ces graphes en tenant compte des étiquettes des contraintes flexibles et de l'étiquette de la contrainte à ajouter ou à retirer pour déterminer les graphes-solutions *GLM*.

Dans cette section, on décrit en détail les procédures généralisées d'ajout et de retrait d'une contrainte c au système de hiérarchie de contraintes. On décrira également quelques propriétés de l'ensemble G (ensemble des graphes admissibles du système). On définira d'abord quelques notations utilisées par ces procédures.

Notations :

On dénote par $\overline{c}$ la paire *(étiquette, poids)* contenant l'étiquette et le poids (ou indice) de la contrainte c, et par M_c l'ensemble des méthodes de la contrainte c. Chaque méthode m dans M_c sera représentée par un couple *(triplet, $\overline{m}$)* où triplet est de la même forme que celui défini dans le chapitre précédent.

On dénote par *gaep* le couple *(ga, ep)* avec *ga* est un *triplet* représentant un graphe-admissible construit par la conjonction (définie au chapitre précédent) des méthodes qui le composent et *ep* une liste de paires *(étiquette, poids)* construite en utilisant l'opérateur $\oplus$[1] et les étiquettes ainsi que les poids des méthodes de ce graphe-solution.

$G = ((g_1,q_1)..... (g_n,q_n))$, l'ensemble G est partitionné en n sous ensembles. Chaque sous ensemble g_i contient les graphes-admissibles ayant la même *liste-réelle-étiquettes* ($g_i = \{gaep_1....gaep_m\}$ telle que $ep_1 = ... = ep_m$).

La *liste-réelle-étiquettes* d'un sous ensemble g_i est noté $\overline{g_i}$ et est égale à la liste des paires *(étiquette, poids)* d'un graphe-admissible dans g_i.

Chaque sous-ensemble g_i est associé à une queue q_i . Si l'on suppose que $q_i = \{c_0, ..., c_n\}$, on dénote par $\overline{q_i}$ la liste des paires *(étiquette, poids)* obtenue par $(\overline{c_0} \oplus ..... \oplus \overline{c_n})$.

La *liste-potentielle-étiquettes* d'un sous ensemble g_i est la liste des paires *(étiquette, poids)* obtenue par $\overline{g_i} \oplus \overline{q_i}$.

1 Cet opérateur sera paramétré par le type de comparateur utilisé par l'utilisateur et aura donc des fonctionnalités différentes

6.3.1 Procédure Ajout-contrainte

Lorsqu'une nouvelle contrainte c est introduite dans le système, la procédure *Ajouter-contrainte* ajoute cette contrainte c dans les queues associées aux sous-ensembles formés dans G, excepté la queue du premier sous ensemble de G (ligne 1; figure 29). Le premier sous-ensemble de G est retiré (ligne 2; figure 29). Ce sous-ensemble retiré est utilisé pour un ajout effectif de la contrainte c sur les différents *gaep* qui le composent (ligne 3; figure 29). La procédure fusionne et ordonne l'union de l'ensemble des couples générés par l'ajout effectif de la contrainte c sur le sous-ensemble g_1 (c.à.d. ce qui résulte de l'appel de la procédure Ajouter et affecté à G'), et les sous-ensembles restant dans G.

Cet ordonnancement est basé en premier lieu sur l'ordre lexicographique de *liste-potentielles-étiquettes* des sous-ensembles, et en second lieu sur liste-réelle-étiquettes de ces sous-ensembles. L'ensemble de ces sous-ensembles ordonné est mis dans G (ligne 4; figure 29).

La procédure examine la queue du sous-ensemble g_1 (g_1 est une variable muette. Elle représente toujours le premier sous-ensemble calculé dans G), dans le cas où cette queue est non vide (ligne 5; figure 29) (c.à.d. on n'est pas certain d'avoir calculé un graphe-solution), la procédure enlève ce premier sous-ensemble de G pour le traiter (ligne 6; figure 29).

La procédure ajoute aux différents *gaep* dans g_1 toutes les contraintes laissées en attente d'ajout dans la queue de ce sous-ensemble. On ajoute d'une manière effective la première contrainte de la queue à l'ensemble des *gaep* de ce sous-ensemble. A l'issue de cet ajout effectif, d'autres *gaep* peuvent être déduits et donc un nouvel ensemble est formé. L'ajout de la deuxième contrainte de la queue se fera sur ce nouvel ensemble. Le processus est le même pour toutes les contraint restantes dans la queue (ligne 7,8; figure 29). L'ensemble des sous-ensembles déduit de cet ajout est fusionné et ensuite ordonné avec l'ensemble des sous-ensembles restant dans G, le résultat est affecté à G (lignes 9; figure 29). La procédure continue de traiter le premier sous-ensemble dans G jusqu'à ce que sa queue devienne vide (ligne 5; figure 29). Cette manœuvre permet de calculer le(s) graphe(s)-solution(s) de la hiérarchie traitée.

La procédure retourne le premier sous-ensemble g_1 de G. Ce premier sous-ensemble contient les meilleurs[1] graphes-admissibles qui sont des graphes-solutions. L'application de l'algorithme de la propagation locale sur ces graphes-solutions produit les valuations qui sont solutions de la hiérarchie.

FIGURE 29 : Procédure d'ajout d'une contrainte au système

Ajouter-contrainte *(c , G, comp)*

1 *Pour chaque* q_i *telle que* $2 \leq i \leq n$ *faire :* $q_i \leftarrow q_i + c$ *Fin-Pour.*
2 $G \leftarrow G \setminus (g_1, q_1)$
3 $G' \leftarrow$ *(Ajouter(c,*g_1*))*
4 $G \leftarrow$ *Trier-Lex (Fusionner (G', G))*
5 *Tantque* $\neg$ *Vide(*q_1*) Faire:*
6 $G \leftarrow G \setminus (g_1, q_1)$
7 $G' \leftarrow g_1$
8 *Pour chaque* $c \in q_1$ *faire :*$G' \leftarrow$ *Ajouter(c, G', comp,G) Fin-Pour.*
9 $G \leftarrow$ *Trier-Lex(Fusionner(G', G))*
Fin-Tantque
10 *Retourner(*g_1*)*

1. Ici meilleur est relatif au comparateur *Comp* (*Comp* $\in \{\tau_{NCNS}, \tau_{CMC}, \tau_{SFP}\}$)

L'ajout effectif d'une contrainte c à un sous-ensemble de couples *gaep* (chaque *gaep* dans g_i est composé d'un graphe-admissible et de la liste construite par les paires *(étiquette, poids)* des méthodes qui composent ce graphe-admissible) est réalisé par la procédure *Ajouter* de la figure 30. Cette procédure tente d'ajouter toutes les méthodes de la contrainte c à tous les *gaep* de g_i (ligne 2,3; figure 30). Chaque ensemble résultant de cet ajout est rangé dans l'ensemble G' qui est de même type que l'ensemble G. Ce rangement consiste à mettre chaque élément d'un ensemble résultant dans le sous-ensemble ayant la même liste-réelle-étiquettes que cet élément. La procédure retourne ces ensembles rangés contenant les nouveaux graphes-admissibles intégrant la contrainte c.

FIGURE 30 : Ajout effectif d'une contrainte sur un ensemble de graphe-admissible

```
Ajouter(c, g_i, comp,G)
1 G'← ∅
2 Pour chaque m ∈ c faire:
3    Ranger( G', Ajouter-méthode(m, g_i, comp,G)))
  Fin-Pour
5 Retourner(G')
```

FIGURE 31 : Ajout effectif d'une méthode sur un ensemble de graphe-admissible

```
Ajouter-méthode(m, g_i, comp, G)
1  G'← ∅
2  G"← ∅
3  Pour chaque gaep ∈ g_i faire:
4        Si Consistant(m, ga) alors Ranger( (ga ∧ m, ep ⊕_comp m̄ ), G')
5              sinon soit M_r ={m' ∈ sg / m̄' =( requise,⊥)}
6                    Si Consistant(m, M_r) alors
7                       G" = {(M, M) ,∅ / (M ⊂ ga) ∧ Consistant(m, M))}
8                       Pour chaque gaep' ∈ G" fair :_
9                         gaep'← (ga' ∧ m, ep' ⊕_comp m)
10                        Si ¬((∃ gaep ∈ g_k /(g_k,∅)∈ G' ∧ gaep' ⊆ gaep)
11                              ∨(∃ gaep ∈ g_j /(g_j,q_j)∈ G ∧ gaep' ⊆ gaep))
12                        alors Ranger((gaep',∅) , G')
                          Fin-Si
                        Fin-Pour
13                    sinon Ranger(gaep, G')
                    Fin-Si
       Fin-Si
    Fin-Pour

14 Retourner(G')
```

La procédure *Ajouter-méthode* effectue un ajout de la méthode m sur les différents graphes-admissibles dans g_i(ligne 3; figure 31). La procédure teste la consistance (définie dans le chapitre précédent) de la méthode avec un graphe-admissible. S'il y a consistance, ce graphe-admissible est modifié en effectuant la conjonction (définie dans le chapitre précédent) de sa représentation et celle de la méthode et en ajoutant la paire *(étiquette, poids)* de la méthode à la liste des paires du graphe-admissible (lignes 4 ; figure 31) (cette ajout est réalisé par l'opérateur $\oplus$. Cet opérateur est paramétré par le comparateur utilisé. On décrira comment cet opérateur procède dans un paragraphe ultérieur).

Dans le cas contraire, la procédure extrait de ce graphe-admissible l'ensemble des méthodes requises et les met dans M_r (qui sont étiquetées par l'étiquette *requise*) (ligne 5; figure 28). Si ce dernier est consistant avec la méthode à ajouter (ligne 6; figure 28) (ceci veut dire qu'il n'y a pas de conflit ni de cycle entre la méthode à ajouter et les méthodes requises du graphe-admissible, et, l'ajout de cette méthode est donc possible), un ensemble de nouveaux graphes-admissibles sera déduit à partir des méthodes consistantes avec la méthode à ajouter (lignes 8; figure 31). L'ensemble $G"$ comporte cet

ensemble déduit de graphes-admissibles. *G"* possède la propriété de ne contenir que des graphes-admissibles maximaux (c.à.d *G"* ne comporte pas deux graphes-admissibles qui se recouvrent).

La méthode *m* est intégrée à chaque graphe-admissible déduit (ligne 9,10; figure 31). Le nouveau graphe-admissible obtenu est rangé dans *G'* s'il est maximal[1] (lignes 10,11,12; figure 31). Si la méthode *m* n'est pas consistante avec une des méthodes requises du graphe-admissible du couple *gaep* alors cette méthode est rejetée et le couple *gaep* est rangé dans *G'* (ligne 13, figure 31). La procédure retourne l'ensemble des sous-ensembles déduits à l'issue de l'ajout de la méthode *m* au sous-ensemble g_i (ligne 14 ; figure 31).

L'opérateur ⊕ est paramétré par le comparateur utilisé pour la résolution de la hiérarchie. Ce comparateur (noté par *Comp* dans les procédures) peut être un des trois comparateurs globaux définis auparavant. Lors de la conjonction d'une méthode avec un ensemble de méthodes, les opérations réalisées par cet opérateur sont comme suit :

Lorsque le comparateur *Nombre-Contraintes-Non-Satisfaites* est utilisé, l'opérateur ⊕ fusionne deux listes ordonnées d'étiquettes en une seule liste ordonnée.

Lorsque le comparateur *Cardinal-Pire-Cas* est utilisé, l'opérateur ⊕ fusionne deux listes ordonnées de paires *(étiquette, (indice,Card(indice)))* en une seule liste en groupant respectivement les couples ayant les mêmes indices et les mêmes étiquettes.

Si le comparateur *Somme-Pondérée-Prédicat* est utilisé alors l'opérateur ⊕ fusionne deux listes ordonnées de paires *(étiquette, poids)* en une seule liste en ajoutant respectivement les poids dans les couples ayant les mêmes étiquettes.

6.3.2 Procédure Retirer-contrainte

Dans le but de retirer la contrainte *c* du système, la procédure *Retirer-contrainte* partitionne l'ensemble *G* en deux ensembles *G'* et *G"* (ligne 1;figure 32). L'ensemble G" est composé des sous-ensembles qui contiennent la contrainte *c* dans leur queue d'attente de contraintes à ajouter. Ceci implique qu'aucune des méthodes de la contrainte *c* ne figure dans les représentations des graphes-admissibles de ces sous-ensembles. Dans ce cas, la procédure de retrait de contrainte retire tout simplement la contrainte *c* de ces queues (puisqu'il n'y a pas eu un ajout effectif de *c* sur les différents graphes-admissibles de ces sous-ensembles) (ligne 2; figure 32).

L'ensemble G' est composé de sous-ensemble ne possédant pas la contrainte c dans leur queue associée. Certains de ces sous-ensembles contiennent dans leurs différentes représentations une des méthodes de la contrainte *c* (lignes 3,4,5; figure 32) (c.à.d. qu'il y a eu un ajout effectif de la contrainte *c* sur les différents graphes-admissibles des couples de ces sous-ensembles. Parmi ces sous-ensembles, certains peuvent parfaitement avoir des queues non vides). Dans ce cas, la procédure de retrait procède de la manière suivante : pour un graphe-admissible qui inclut dans sa représentation une des méthodes de la contrainte *c* (ligne 5; figure 32), ce graphe-admissible sera enlevé du sous-ensemble qui le contient (ligne 6; figure 32). La méthode de la contrainte *c* sera retirée de ce graphe-admissible en utilisant l'opérateur — et une nouvelle représentation sera déduite (ligne 7; figure 32). Cette dernière est prise en compte s'il n'existe aucune autre représentation dans les sous-ensembles de *G'* ou ceux de *G"* qui la recouvre, ceci pour garder la maximalité des graphes-admissibles (ce

1. Bien que ce graphe-admissible soit maximal par rapport à l'ensemble G", il peut ne pas être maximal par rapport à l'ensemble G' (ou à l'ensemble *G*) comme le montre l'exemple suivant : - hypothèses: seule la méthode m_1 est requise, $g_i=\{(m_1 \wedge m_2,\ \overline{m}_1 \oplus \overline{m}_2),(m_1 \wedge m_3,\ \overline{m}_1 \oplus \overline{m}_3)\}$, ¬consistante($m_3$, m_4) , consistante(m_1 , m_4), consistante(m_2 , m_4) et on veut ajouter la méthode m_4 aux gaep de g_i. Au pas 4 de l'algorithme, l'examen du premier couple de g_i résulte en $G' = \{(m_1 \wedge m_2 \wedge m_4,\ \overline{m}_1 \oplus \overline{m}_2 \oplus \overline{m}_4), \emptyset)\}$. L'examen du deuxième couple au pas 7 résulte en $G''=\{(m_1,\ \overline{m}_1), \emptyset)\}$ au pas 9 : $gaep'=(m_1 \wedge m_4,\ \overline{m}_1 \oplus \overline{m}_4)$. On voit bien que le couple calculé est recouvert par le couple dans *G'* .

nouveau *gaep* déduit ne sera plus un élément de g_i puisqu'il ne possède pas la même *liste-réelle-étiquettes* que celles des éléments restants dans g_i. Un nouveau sous-ensemble contenant ce *gaep* sera construit et la queue q_i lui sera associée. Ce nouveau sous-ensemble sera rangé dans *G")* (ligne 8; figure 32).

Les ensembles *G'* et *G"* sont fusionnés et le résultat est ordonné selon la *liste-potentielle-étiquettes* des sous-ensembles qui la composent (cette opération consiste à fusionner les sous-ensembles ayant la même queue de contraintes à ajouter et la même *liste-réelle-étiquette* en un seul sous-ensemble)(ligne 9; figure 32).

Après cet ordonnancement, il se peut que la queue des contraintes à ajouter du premier sous-ensemble dans *G* soit non vide (ligne 10; figure 32). Dans ce cas, l'opération d'ajout effectif des contraintes de cette queue sur les couples de ce sous-ensemble est déclenchée (lignes 11,12,13,14; figure 32). La procédure s'achève lorsque la queue du premier sous-ensemble dans *G* est vide (c.à.d. *liste-potentielle-étiquettes*(g_1) = *liste-réelle-étiquettes*(g_1)). La procédure retourne le premier sous-ensemble dans *G* (ligne 15; figure 32). Celui-ci contient les graphes-solutions *GLM* du système de contraintes hiérarchiques.

FIGURE 32 : Procédure de retrait d'une contrainte

Retirer-contrainte(c , comp,G)

```
1  Soit G' = {(g_i, q_i) / (c ∉ q_i)} et G'' = {(g_i, q_i) / (c ∈ q_i)}
2  Pour chaque (s_i, q_i) ∈ G'' faire : (q_i ← q_i \ c) Fin-Pour
3  Pour chaque (s_i, q_i) ∈ G' faire :
4    Pour chaque gaep ∈ g_i faire:
5      Si (∃ m ∈ c ∧ m ∈ ga) alors
6        g_i ← g_i \ gaep
7        gaep ← (ga — m, ep- m)
8        Si ¬(∃ gaep' ∈ g_i / ((g_i, q_i) ∈ G' ∨ ∈ G'' ) ∧ gaep ⊆ gaep') alors Ranger( (gaep,q_i), G'')Fin-Pour
       Fin-Si
     Fin-Pour
   Fin-Pour
9  G ← Trier-Lex(Fusionner(G', G''))
10 Tant que ¬ Vide(q_1) Faire:
11     G ← G\(g_1,q_1)
12     G'← g_1
13     Pour chaque c ∈ q_1 faire :G' ← Ajouter(c, G', comp,G) Fin-Pour.
14     G ← Trier-Lex(Fusionner( G', G))
   Fin-Tant que
15 Retourner(g_1)
```

6.3.3 Quelques propriétés et caractéristiques de l'ensemble *G*

Dans cette section, on caractérise d'une façon formelle cet ensemble *G* de graphes-admissibles, dans le but de clarifier la structure de cet ensemble. On imposera des axiomes qui nous serviront à prouver des théorèmes. Ces derniers nous permettront d'obtenir des propriétés sur cet ensemble *G*.

Axiome 6.1 :

$$\forall gaep \in G\ \forall m \in ga \neg(\exists m' \in ga\ /\ m = m')$$

Une méthode d'une contrainte quelconque du système existe au plus une seule fois dans un graphe-admissible de *G*.

Axiome 6.2 :

$Soit(m,m') \in (c,c')(\bar{c} \neq \bar{c}' \Rightarrow m \neq m')$

Deux contraintes différentes ont des méthodes différentes.

Axiome 6.3 :

$\neg(\exists gaep \in G\ \exists gaep' \in G\ /\ (gaep = gaep'))$

L'ensemble G ne contient pas de graphes-admissibles redondants.

Théorème 6.1:

Pour un des trois comparateurs utilisés, si un graphe ga' dans G est plus (ou aussi) important qu'un autre graphe ga dans G alors, ga' n'est pas un sous-graphe de ga.

$(\forall Comp \in \{NCNS, CPC, SPP\})(\forall gaep,gaep' \in G)ep \leq_{Lex} ep' \Rightarrow gaep' \not\subset gaep$

Preuve:

d'après l'axiome 3, on peut déduire que : $ep' \leq_{Lex} ep \Leftrightarrow \exists m \in ga \exists m' \in ga'\ /\ \bar{m}' \neq \bar{m}$ et d'après l'axiome 2, on a $m \neq m'$ et donc : $gaep' \not\subset gaep$.

Proposition 6.3:

Si une méthode est non consistante avec un sous-graphe d'un graphe-admissible, alors elle est non consistante avec ce graphe-admissible.

$(\forall ga)(\forall ga)(\forall m)ga' \subset ga \wedge \neg\text{Consistante}(m, ga') \Rightarrow \neg\text{Consistante}(m, ga)$

Théorème 6.2:

Pour un des trois comparateurs utilisés, si un graphe ga dans G est moins (ou aussi) important qu'un autre graphe ga' dans G alors, ga n'est pas un sous-graphe de ga'.

$(\forall Comp \in \{NCNS, CMC, SPP\})(\forall gaep,gaep' \in G)ep \leq_{Lex} ep' \Rightarrow gaep \not\subset gaep'$

Preuve:

On distingue trois cas :

- Si $|ga'| > |ga| \Rightarrow \exists m \notin ga \wedge m \in ga'$ et donc : $ga' \not\subset ga$

- Si $|ga'| = |ga| \Rightarrow$ par axiome 3 : $\exists m \in ga' \exists m' \in ga$ telle que $\bar{m} \neq \bar{m}'$

Or d'après l'axiome 1 on a : $m \neq m'$ et donc $ga' \not\subset ga$

- Si $|ga'| < |ga|$ on montre que $ga' \not\subset ga$ par l'absurde :

supposons que : $ga' \subset ga$, ceci implique la véracité des propositions suivantes :

- $\forall m' \in ga' \Rightarrow m' \in ga$ (puisqu'on a supposé l'inclusion)

- $\exists m \in ga$ telle que$(m \notin ga') \wedge$ Consistante$(m, ga - m)$ (puisqu'on a l'inclusion stricte)

- $\neg$Consistante(m, ga') (puisque si m était consistante avec ga', on l'aurait déjà ajoutée à ga', car par construction un graphe-admissible est maximal).

Or d'après la proposition on a: $\neg$Consistante(m, ga) et d'après l'hypothèse on a: Consistante$(m, ga - m)$ ce qui est une contradiction.

Propriété 6.1:

$$\forall gaep,gaep' \in G\ (ga - ga' \neq \varnothing \wedge ga' - ga \neq \varnothing)$$

Deux graphes-admissibles quelconques de G sont distincts.

Propriété 6.2:

$$\text{Soit } gaep \in G \text{ et } c \in H_l(\forall m \in c, m \not\subset gaep \Rightarrow \neg \text{Consistant}(c,gaep))$$

Chaque graphe-admissible dans G est maximal et consistant.

6.4 Complexité, implémentation et mesures de performance

Sannella dans [San94] prouve que la classe des problèmes concernant le comparateur *Nombre-Contraintes-Non-Satisfaites* est N-P complet. Etant donné que les deux autres comparateurs utilisés ici (*Somme-Pondérée-Prédicat* et *Cardinal-Pire-Cas*) sont plus complexes que le premier, on a tendance à croire fortement que ces problèmes sont aussi N-P complets. Dans ces conditions, il est clair que dans le pire de cas, Houria est exponentiel. Cependant, les techniques d'optimisation et l'heuristique utilisées permettent de réduire les opérations afin de converger vers une solution en un temps acceptable.

Les algorithmes généralisés et les définitions que nous avons présentés auparavant ont été programmés avec Le-Lisp version 15.25 [Ilg87]. Houria produit des graphes-solutions et leur applique la propagation locale. Dans le but de tester les performances de ce résolveur, nous l'avons expérimenté sur des problèmes sur-contraints générés aléatoirement.

Chaque problème généré est basé sur trois paramètres : le nombre *nv* de variables, le nombre *nc* de contraintes et l'arité de la contrainte *ac*. Chacune des contraintes générée est étiquetée par une étiquette générée aléatoirement parmi l'ensemble des étiquettes *{requise, forte, moyenne, faible}* et pondérée par un poids numérique généré aléatoirement dans l'ensemble fini *{0.1, 0.2, ... 1}*[1]. Parmi les critères intégrés, nous avons exécuté trois types de tests décrits dans les sections suivantes :

6.4.1 Tests avec arité fixe

Le premier type est composé de deux séries de tests :

La première série consiste en des contraintes d'arité égale à 3 (c.à.d. *ac=3*, ici on fixe le nombre de variables conséquentes de chaque méthode de la contrainte à 1. Le nombre de méthodes dans chaque contrainte générée est un nombre tiré aléatoirement entre 1 et 3, ceci est proche des applications réelles puisqu'on suppose qu'on n'a pas toujours des contraintes régulières).

La deuxième série consiste en des contraintes d'arité égale à 4 (c.à.d. *ac=4*, ici les méthodes des contraintes générées sont des méthodes dont le nombres des variables conséquentes est de 1 ou 2. Le nombre de méthodes de chaque contrainte générée dépend du nombre de variable conséquentes de la contrainte. Dans le cas où ce nombre est égal à 1 alors le nombre de méthodes dans la contrainte est un nombre tiré aléatoirement entre 1 et 4, tandis que si ce nombre est égale à 2 alors le nombre de méthodes dans la contrainte est un nombre tiré aléatoirement entre 1 et 6).

1. Excepté pour les tests utilisant le critère *Nombre-Contraintes-Non-Satisfaites* où le poids est 1 pour toutes les contraintes générées.

Pour chacune des séries, nous avons mesuré le temps mis par Houria pour planifier un graphe solution. Nous avons répété les mesures pour 50 problèmes différents sur-contraints générés et nous avons calculé le temps moyen de ces mesures. Les résultats obtenus en utilisant les critères *Nombre-Contraintes-Non-Satisfaites*, *Cardinal-Pire-Cas* sont respectivement dans les figures 33 et 34. L'axe des abscisse représente le nombre de contrainte introduites, et l'axe des ordonnés est en échelle logarithmique et représente le temps mis pour planifier un graphe solution. Les différentes mesures des figures 33 et 34 peuvent être interprétées comme suit :

Pour un nombre donné de variables *nv*, le nombre de contrainte générées est faible par rapport au nombre total qui peut être généré. Par exemple, considérons les contrainte d'arité 4, pour *nv=30* le nombre de méthodes différentes qu'on peut générer est de $(4+6)\times C_{30}^{4} = 274050$. Si l'on suppose qu'en moyenne il y a 5 méthodes par contrainte, on aura 54810 contraintes différentes et donc, 100 contraintes générées représentent une "densité" de 0.002.

Pour un nombre de contraintes donné, plus le nombre de variables est grand moins le temps mis est important. Ceci est parfaitement normal, puisque le tirage des contraintes se fait d'une façon aléatoire et on a plus de chance d'avoir moins de conflits et moins de cycles lorsqu'on considère un ensemble de variables plus important à nombre de contraintes fixé.

Le temps mis en considérant des contraintes d'arité 4 est supérieur à celui mis en considérant des contraintes d'arité 3. Ceci peut s'expliquer par le fait qu'avec les contrainte d'arité 4, on traite plus de conflits et plus de circuits qu'avec les contraintes d'arité 3.

Avec *ca* =3 ou *ca* = 4, le temps mis en utilisant le critère *Nombre-Contraintes-Non-Satisfaites* est strictement inférieur à celui mis par le critère *Cardinal-Pire-Cas*. Ceci peut s'expliquer par le fait suivant :

on effectue plus d'opérations avec ce deuxième critère qu'avec le premier. En effet, puisqu'avec le deuxième critère les contraintes sont étiquetées et possèdent des indices, alors chaque contrainte tirée a plus de chance de déclencher l'examen d'un nombre plus important d'éléments dans *G* que si la contrainte était uniquement étiquetée. Ceci est dû à la définition de ce deuxième critère : on considére qu'une solution est aussi bonne qu'une autre si toutes les deux satisfont le même nombre de contraintes associées au plus fort indice sans se préoccuper des contraintes de remplacement (celles associées à des indices moins forte). Dans ces conditions, deux (ou plusieurs) graphes de méthodes actives feront partie d'un même sous-ensemble (possèdent la même *liste-réelle-étiquettes*). L'ajout d'une nouvelle contrainte se fera d'une façon effective sur tous les graphes de cet ensemble. Par conséquent le temps de traitement devient plus important. Par exemple, considérons les trois graphes de méthodes ga_1, ga_2 et ga_3 telle que :

ga_1 active la contrainte c_1 étiquetée par *forte* et associée à l'indice 0.9,

ga_2 active les deux contraintes c_2 et c_3 étiquetées par *forte* et associées respectivement aux indices 0.9 et 0.8,

ga_3 active les trois contraintes c_4, c_5 et c_6 étiquetées par *forte* et associées respectivement aux indices 0.9 et 0.8 et 0.7,

Ces trois graphes de méthodes ferons partie d'un même sous-ensemble *g* de *G*. Ce sous-ensemble aura la liste *Liste-réelle-étiquettes* égale à *(forte, (0.9,1))*. L'ajout d'une nouvelle contrainte sur ce sous-ensemble se fera sur les trois graphes-solutions dans *g*.

Tandis que si l'on considère le critère *Nombre-Contraintes-Non-Satisfaites* et on suppose également qu'on a les trois graphes de méthodes ga_1, ga_2 et ga_3 et que les contraintes sont uniquement étiquetées alors, on aura trois sous-ensembles g_1, g_2 et g_3 ordonnés dans *G* telle que $g_1=\{ga_3\}$, $g_2=\{ga_2\}$ et $g_3=\{ga_1\}$. L'ajout d'une nouvelle contrainte se fera en

priorité sur g_1 ensuite si on n'est pas certain d'avoir calculé le meilleur graphe, alors on continu sur g_2 etc.

Par le critère *Nombre-Contraintes-Non-Satisfaites* (resp. *Cardinal-Pire-Cas)*, lorsqu'en moyenne le nombre de contraintes est trois fois (resp. 2.2 fois) supérieur au nombre de variables, le temps mis (entre 0 et 100 secondes) est acceptable pour certaines applications graphiques. Mais lorsque le nombre de contraintes est plus que 3 fois (resp. 2.2 fois) le nombre des variables, le temps mis est très long. Par exemple, dans la figure 34.b, en considérant *nv=30* les temps, en secondes, respectifs pour traiter 60, 80 et 100 contraintes sont: 51, 125, 368.

FIGURE 33 : Temps d'exécution avec le critère NCNS lorsque ac=3, ac=4 et nv varie.

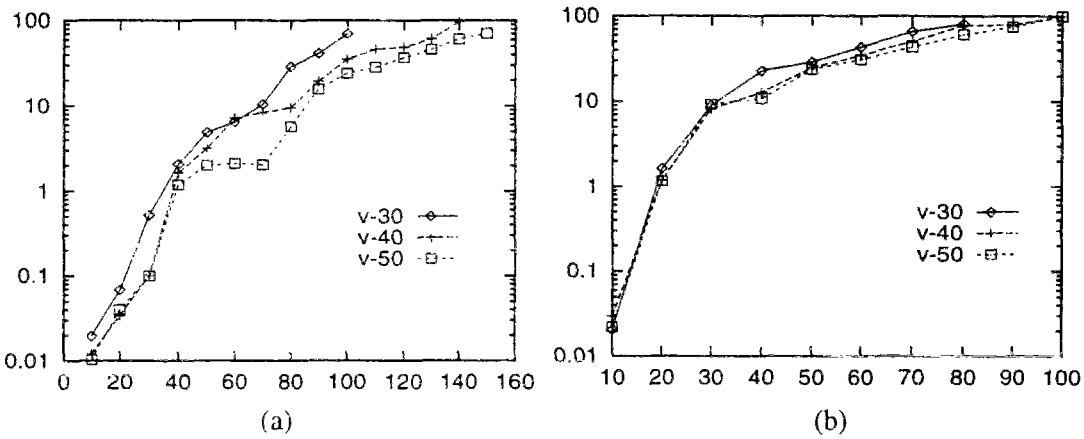
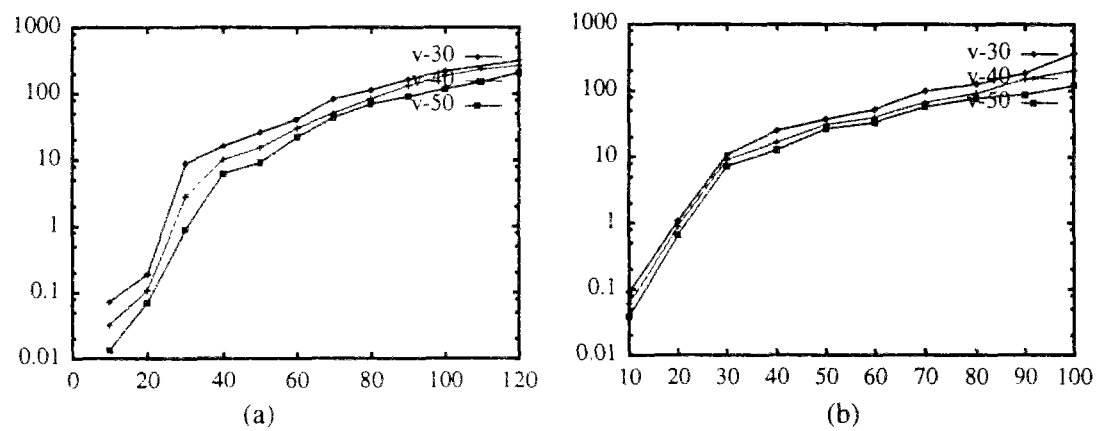


FIGURE 34 : Temps d'exécution avec le critère CPC lorsque ac=3, ac=4 et nv varie



6.4.2 Tests avec arité variable

Dans ce deuxième type de tests, le nombre de contraintes varie de 20 à 200, par pas de 20. Pour ce type de test, les contraintes générées sont d'arités 2, 3 ou 4. 70% des contraintes générées sont d'arité 2, 20% d'arité 3 et 10% d'arité 4. Ici on génére des hiérarchies de contraintes qui sont plus proche de hiérarchies d'applications réelles. On suppose également que :

pour les critères *Nombre-Contraintes-Non-Satisfaites* et *Somme-Pondérée-Prédicat* , le nombre de contraintes est trois fois supérieur au nombre de variables *nv*.

Les résultats obtenus en utilisant ces deux critères apparaissent respectivement dans les figures 35 et 36.

FIGURE 35 : *Temps d'exécution avec le critère NCNS*

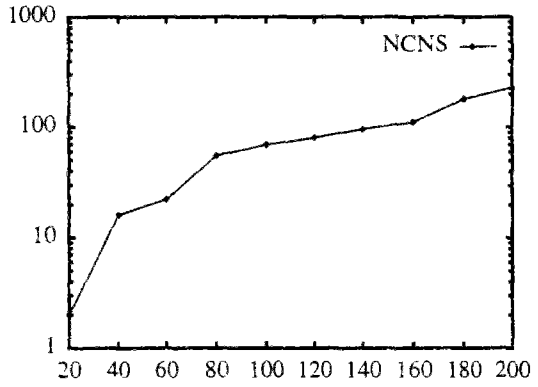
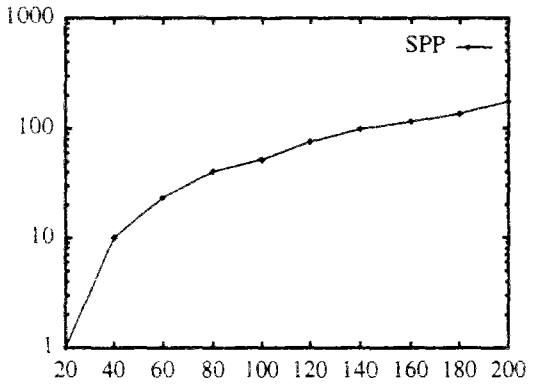


FIGURE 36 : *Temps d'exécution avec le critère SPP*



Ces différentes mesures peuvent être interprétées comme suit :

le temps mis en considérant un ensemble de contraintes contenant 70% de contrainte d'arité 2, 20% de contraintes d'arité 3 et 10% de contraintes d'arité 4 est inférieur celui mis en considérant uniquement des contraintes d'arité 3 ou 4. Ceci peut s'expliquer par le fait que la majorité des contraintes générées sont binaires et donc :

on a plus de chance de traiter un nombre inférieur de conflits et de circuits que celui obtenu si on ne manipulait que des contraintes d'arité 3 ou 4,

aussi, on a plus de chance d'utiliser les théorèmes 5.2 et 5.3 pour prédire l'absence ou l'existance de circuit dans un graphe solution.

le temps mis en utilisant le critère *Nombre-Contraintes-Non-Satisfaites* est légèrement supérieur à celui mis par le critère *Somme-Pondérée-Prédicat.* Ceci peut s'expliquer par les faits suivants :

le critère *Nombre-Contraintes-Non-Satisfaites* a le même comportement, au sens de l'implémentation, que le critère *Somme-Pondérée-Prédicat* puisque si toutes les contraintes sont pondérées par le poids 1, alors ces deux critères sont équivalents,

pour le critère *Somme-Pondérée-Prédicat* les contraintes sont pondérées par un poids tiré aléatoirement de *{0.1,...,0.9}* et donc lors de l'ajout d'une contrainte on a plus de chance de faire un nombre moins important d'opérations pour trouver un graphe solution puisque ici chaque élément de G comporte en moyenne un graphe-admissible (contrairement à lorsqu'on utilisait le critère *Cardinal-Pire-Cas*).

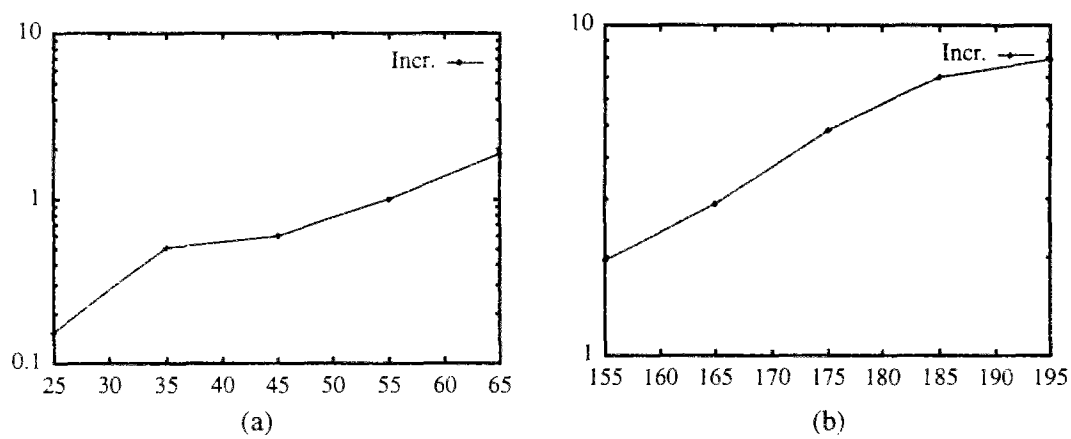
6.4.3 Tests d'incrémentalité

Le troisième type de tests consiste à tester l'aspect incrémental de ce résolveur. Ici, on suppose que le nombre de variable *nv* est le tiers du nombre de contraintes *nc* générées et que les contraintes générées ont les mêmes caractéristiques que celle du deuxiéme type de tests. Le nombre de contraintes varie de 10 à 200, par pas de 10. On calcule le temps nécessaire pour l'ajout de la $25^{ème}$ contrainte (resp. de la $35^{ème}$ contrainte, ..., et de la $195^{ème}$) générée. L'expérience est faite 50 fois et le temps moyen est reporté sur la figure. Les résultats obtenus en utilisant le critère *Nombre-Contraintes-Non-Satisfaites* sont ceux de la figure 37.

Ces différentes mesures peuvent être interprétées comme suit :

plus le nombre de contraintes dans le système est important plus le temps mis pour l'ajout d'une contrainte est important, ceci est dû au nombre d'éléments dans G qui croît vite par rapport au nombre de contraintes introduite au système et par conséquent, d'un ajout à un autre, on est amené à examiner plus d'éléments de G pour trouver une solution.

FIGURE 37 : Temps d'incrémentalité pour le critère NCNS



6.5 Comparaison fonctionnelle avec d'autres résolveurs

Comme il est mentionné au chapitre 4, SkyBlue est le successeur de DeltaBlue. Comme QuickPlan et Deltablue, SkyBlue est un résolveur de contraintes fonctionnelles d'un système hiérarchique. QuickPlan, DeltaBlue et SkyBlue manipulent des contraintes fonctionnelles. Chacune de ces contraintes peut avoir plusieurs méthodes. Chaque méthode peut avoir plusieurs variables en entrée et plusieurs en sortie. QuickPlan, DeltaBlue et SkyBlue utilisent la propagation locale basée sur le comparateur *Localement-Prédicat-Meilleur*, et construisent des graphes localement meilleurs (GLM). SkyBlue supporte les graphes de contraintes cycliques et fait appel à un résolveur qui satis-

fait ces contraintes simultanément. Les principales différences entre Houria, QuickPlan, SkyBlue et DeltaBlue sont les suivantes :

- Skyblue traite les contraintes cycliques (en faisant appel à un résolveur de cycle) et les contraintes possédant plusieurs méthodes, tandis que QuickPlan, DeltaBlue et Houria traitent les contraintes possédant plusieurs méthodes et éliminent les cycles. Houria peut être étendu pour manipuler les cycles, il suffit de prévoir un résolveur de cycle et lors de la détection d'un cycle par la définition *Pas-De-Cycle*, faire appel à ce résolveur de cycle (qui résout les contraintes simultanément).

- Etant donné un système hiérarchique de contraintes, Houria manipule plusieurs graphes solutions (ceci est particulièrement intéressant, si l'utilisateur désire avoir plus d'une solution lors de l'ajout ou du retrait d'une contrainte) tandis que QuickPlan, DeltaBlue et SkyBlue manipulent un seul graphe solution qui est localement meilleur.

- Pour un problème sur-contraint, les critères de comparaison utilisés par Houria sont plus fins[1] et donnent des solutions ayant une qualité meilleure[2] que celles produites par le critère utilisé dans QuickPlan, SkyBlue ou DeltaBlue.

- Si la hiérarchie de contraintes possède une ou plusieurs solutions cycliques et au moins une solution acyclique, alors Houria et QuickPlan garantissent de trouver cette solution acyclique, tandis que SkyBlue est incapable de garantir cette solution.

- Le mode de représentation des contraintes utilisé par Houria est souple : il permet de manipuler les graphes solutions comme des ensembles de variables, ce qui rend les calculs moins coûteux en temps et en espace mémoire. L'intégration dans Houria de n'importe quel critère global de comparaison basé sur l'erreur prédicat est possible. Il suffit d'attribuer des nouvelles fonctionnalités à l'opérateur $\oplus$. Tandis que l'intégration d'un nouveau critère local ou global à l'un des résolveurs suivant QuickPlan, DeltaBlue ou SkyBlue nécessiterait la modification des noyaux de ces derniers.

- Houria manipule des hiérarchies de contraintes où les contraintes d'un niveau donné peuvent être pondérées par des poids numériques (ou indices de satisfaction) tandis que QuickPlan, DeltaBlue et SkyBlue manipulent des hiérarchies de contraintes où les contraintes sont uniquement étiquetées.

L'exemple suivant illustre la comparaison entre un des critères utilisés par Houria (c.à.d *Nombre-Contraintes-Non-Satisfaites*) et celui utilisé par SkyBlue (c.à.d *Localement-Prédicat-Meilleur*).

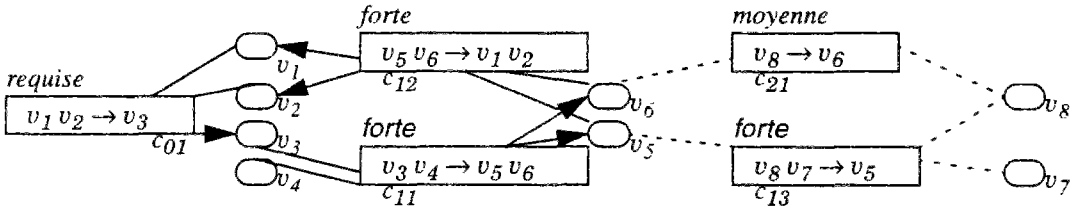
$H = \{ H_0, H_1, H_2 \}, H_0 = \{ c_{01} \}, H_1 = \{c_{11}, c_{12}, c_{13}\}, H_2 = \{c_{21}\}, M_{c_{01}} \equiv \{v_1 v_2 \rightarrow v_3\}$,

$M_{c_{11}} \equiv \{v_3 v_4 \rightarrow v_5 v_6\}$, $M_{c_{12}} \equiv \{v_5 v_6 \rightarrow v_1 v_2\}$, $M_{c_{13}} \equiv \{v_7 v_8 \rightarrow v_5\}$, $M_{c_{21}} \equiv \{v_8 \rightarrow v_6\}$.

Le graphe localement meilleur construit par SkyBlue est celui de la figure 38. *SkyBlue* garde la contrainte c_{13} non active, puisqu'il considère que même si elle devient active, elle ne produirait pas un graphe localement meilleur que celui de la figure 38. L'invariant en SkyBlue est qu'il n'existe pas de contrainte inactive qui puisse être activée en désactivant une ou plusieurs contraintes moins importantes. Aussi lors de l'activation d'une contrainte, il n'y a pas de conflit entre la méthode sélectionnée de cette contrainte et celle d'une autre contrainte plus importante active dans le graphe. SkyBlue garde la contrainte c_{21} inactive, puisqu'elle est en conflit avec la contrainte c_{11}, et elle est moins importante que cette dernière (c.à.d. c_{11} est étiquetée par *forte* tandis que c_{21} est étiquetée par *moyenne*).

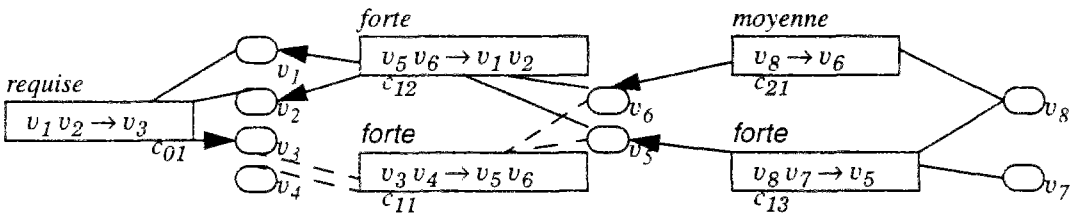
1. Dans le sens où ces critères discriminent mieux l'ensemble des valuations dans S_0 qu'un comparateur local.
2. Pour les critères *Nombre-Contraintes-Non-Satisfaites / Localement-Prédicat-Meilleur*, ces solutions sont meilleures du point de vue du nombre de contraintes satisfaites en respectant la hiérarchie

FIGURE 38 : Graphe localement meilleur de méthodes généré par SkyBlue.



Le graphe lexicographiquement meilleur construit par Houria est celui de la figure 39. La valuation produite par ce graphe doit être préférée à celle produite par le graphe localement meilleur de la figure 38, puisqu'elle est d'une qualité meilleure.

FIGURE 39 : Graphe Lexicographiquement meilleur généré par Houria.



6.6 Perspective

Pour remédier au problème de stockage des états et des tests de maximalité entre ces états, on peut procéder de la manière suivante : la gestion des graphes solutions sera effectuée par un arbre où les noeuds sont des contraintes. Chaque noeud a autant d'arcs que de méthodes plus un arc pour exprimer l'absence de cette contrainte dans la branche. Chaque chemin de la racine à une feuille correspond à un graphe de méthode. Seule la feuille de la branche courante (la branche la plus prometteuse) sera calculée en utilisant l'opérateur de conjonction sur les différentes méthodes qui sont les arcs de ce chemin. L'ensemble G aura la même structure que celle décrite auparavant à savoir un ensemble d'états où chacun de ces états possède une queue. Ici un état est un paquet de noeuds de l'arbre et la queue d'un état renferme les noeuds à ajouter (des contraintes à ajouter). La fonction d'évaluation reste la même.

Lors de l'ajout d'une contrainte, un nouveau noeud va être ajouté à toutes les queues dans G et au chemin courant dans l'arbre. S'il y a consistance entre ce chemin courant et le noeud ajouté, une nouvelle feuille sera calculée et constituera le graphe solution. s'il n'y a pas consistance, il faut remonter de la feuille du chemin courant jusqu'à l'arc le plus haut qui crée cette inconsistance. Ensuite il faut copier cette branche courante en une ou plusieurs sous branches maximales consistantes avec le noeud ajouté. En effet, s'il s'agit d'un conflit de méthodes, seule une seule branche sera ajoutée à l'arbre, mais dans le cas d'un circuit, plusieurs branches peuvent être déduites et ajoutées à l'arbre. La feuille du chemin courant n'est pas stockée. L'ensemble G est ensuite trié selon la fonction d'évaluation f et la branche prometteuse est déterminée. Ce processus continue jusqu'à ce que la queue du premier paquet soit vide.

Le mécanisme pour réaliser cette perspective reste le même que celui décrit auparavant, seulement ici on espère que le fait de ne garder que des références dans G et non des graphes solutions (quitte à les recalculer plusieurs fois lors d'une inconsistance ou lors de changement de branche) nous permet de réduire la complexité en espace mémoire.

Synthèse du chapitre

L'objet de ce sixième chapitre a été l'extension du nouvel algorithme incrémental pour la résolution de hiérarchies de contraintes fonctionnelles conçu au chapitre 5. Cette extension consiste à surmonter la restriction imposée par les résolveurs existants et de prendre en compte des hiérarchies contenant des contraintes étiquetées et pondérées .

D'une façon théorique, si la hiérarchie possède une ou plusieurs solutions acycliques (c.à.d. il existe une ou plusieurs valuations dans S telles que ces valuations satisfassent des ensembles de contraintes ne formant pas de cycle) alors on garantit de trouver ces solutions à partir de ces graphes-solutions en leur appliquant l'algorithme de propagation locale. Maintenant, il se peut que la hiérarchie ne possède pas de solution acyclique et possède uniquement des solutions cycliques, vue la première condition de la définition d'un graphe solution, ces graphes solutions garantissent de trouver les solutions les plus proches de celles de S. Il est à signaler ici qu'on peut aussi trouver ces solutions cycliques, il suffit d'avoir un outil (résolveur de cycle) qui résout un ensemble de contraintes formant un cycle lors de sa détection par le prédicat *Pas-De-Circuit-Bis* (défini au chapitre 5).

Les deux critères de comparaison intégrés à cet algorithme dans ce chapitre sont le comparateur *Cardinal-Pire-Cas* et *Somme-Pondérée-Prédicat*. Le comparateur *Cardinal-Pire-Cas* peut être considéré lorsque l'utilisateur désire exprimer au sein d'une même classe des contraintes de remplacement ou encore des priorités entres les méthodes des contraintes. Par exemple, si l'on considère la contrainte *Somme* $=B+C+D$. Cette contrainte peut avoir 4 méthodes où chacune calcule la valeur d'une des quatre variables à partir des trois autres. Chacune de ces méthodes peut être invoquée pour satisfaire cette contrainte. Si lors d'une modification d'une des valeurs des variable B, C ou D alors il semblerait plus naturel de répercuter cette modification sur la valeur de la variable *Somme* et non sur celle de l'une des variables non modifiées. Pour réaliser ce désir, Il suffit de pondérer la méthode calculant la variable *Somme* par un indice fort, ensuite les méthodes calculant les valeurs des autres variables par un indice moins fort.

Le comparateur *Somme-Pondérée-Prédicat* peut être considéré lorsque le poids d'une contrainte traduit son degré (c.à.d. mesure) d'importance dans la classe de la hiérarchie où elle se trouve si elle est satisfaite.

On montre également que les ensembles de solutions obtenus par ces comparateurs sont différents les uns des autres. Ces ensembles sont aussi différents de celui produit par le comparateur *Nombre-Contraintes-Non-Satisfaites* après subdivision de la hiérarchie.

Le résolveur tel qu'il est conçu au chapitre 5 a permis l'intégration des deux critères globaux décrits dans ce chapitre sans avoir à modifier son noyau. Il a fallu tout simplement attribuer des fonctionnalités différentes à l'opérateur $\oplus$ (à la fonction d'évaluation) selon le comparateur utilisé. Généralement, l'intégration de n'importe quel critère global utilisant l'erreur prédicat (qui retourne 1 si la contrainte n'est pas satisfaite et 0 sinon) dans ce résolveur est possible. Il suffit d'associer sa fonction d'agrégation d'erreur par niveau à l'opérateur $\oplus$.

7 L'utilisation de Houria dans la programmation logique par hiérarchie de contraintes.

Les contraintes fonctionnelles sont utilisées dans plusieurs applications gérées par des langages de programmation logique par contraintes comme par exemple dans les interfaces graphiques, pour des simulations physiques ou encore des contraintes géométriques. Dans plusieurs situations, il est nécessaire de pouvoir exprimer dans ces langages des contraintes dures qui doivent être satisfaites et les contraintes de préférences à satisfaire au mieux selon le comparateur utilisé.

La Programmation Logique par Hiérarchie de Contraintes (*PLHC*) étend la Programmation Logique par Contraintes (*PLC*) en incluant les hiérarchies de contraintes. La *PLC* comme la *PLHC* sont paramétrées par *D* qui est le domaine des contraintes. De plus, la *PLHC* est paramétrée par le comparateur *C* utilisé. Des prototypes de la *PLHC(D,LPM)* sont décrit dans [Wil92, WB89, BMM+89, BDF+87] où *D* est l'ensemble des réels et *LPM* est le comparateur *Localement-Prédicat-Meilleur*. Comme il est mentionné par les auteurs de ces prototypes, l'expérience d'écrire des programmes en *PLHC(R,LPM)* montre que le comparateur *Localement-Prédicat-Meilleur* produit des solutions non intuitives. Ceci résulte du fait que le comparateur *Localement-Prédicat-Meilleur* ne peut comparer que des solutions résultant d'une seule hiérarchie (ceci est appelé comparaison intra-hiérarchie). Cependant, plusieurs applications pratiques en *PLHC* suggèrent le besoin d'une comparaison entre les solutions résultant des différents choix de règles du programme afin d'éliminer celles qui sont non intuitives (c.à.d. jugées moins bonnes) (ceci est appelé comparaison inter-hiérarchies).

Pour réaliser une comparaison inter-hiérarchies, seuls les comparateurs globaux peuvent être utilisés. Le résolveur décrit dans les chapitres 5 et 6 implémente des comparateurs globaux et il peut donc être utilisé à cet effet.

Ce chapitre décrit un algorithme basé sur notre résolveur présenté dans les chapitres 5 et 6 et sur le fondement théorique de comparaison inter-hiérarchies que nous rappellerons dans ce chapitre. Ce fondement est décrit en détail dans [Wil92, WB89], il étend la définition de l'ensemble de solutions afin de donner la possibilité de pouvoir comparer deux solutions résultant de différentes hiérarchies.

Ce chapitre est composé de quatre parties. La première partie est un état de l'art ou nous présenterons l'intégration des hiérarchies de contraintes dans les langages de programmation logique. Il s'agira tout d'abord de décrire globalement la programmation logique par contraintes, ensuite la programmation logique par hiérarchie de contraintes ainsi que le schéma général étendu du comparateur *Globalement-Meilleur* utilisé pour la comparaison inter-hiérarchies. On montre *via* deux exemples le besoin de comparaison inter-hiérarchies. On discutera sur l'aspect des comparateurs, sur les contraintes non primitives et enfin sur les travaux existants dans ce domaine. Ces travaux avaient pour objectif d'obtenir les mêmes résultats que la comparaison inter-hiérarchies. L'exposé de cette partie ne se veut pas exhaustif mais ciblé sur la description des points théoriques auxquels nous ferons référence ultérieurement dans ce chapitre. Dans la deuxième partie, on présente les idées sur lesquelles est basé un nouvel algorithme pour la réalisation de cette comparaison inter-hiérarchies. La troisième partie décrit cet algorithme. Enfin, la quatrième partie décrit un exemple pour bien illustrer cet algorithme.

7.1 Hiérarchie de contraintes et programmation logique

Initialement, les hiérarchies de contraintes ont été une extension du système Thinglab [BDF+87]. Par rapport aux hiérarchies de contraintes et à la programmation logique, nous avons vu que la théorie des hiérarchies de contraintes permet à l'utilisateur de spécifier non seulement les contraintes dures mais aussi les contraintes de préférences réparties en un ou plusieurs niveaux. Ce schéma de hiérarchie de contraintes est paramétré par un comparateur C qui permet la comparaison des différentes solutions possibles d'une seule hiérarchie et le choix de la meilleure solution. Récemment, le paradigme des hiérarchies de contraintes a été intégré avec le schéma de programmation logique par contraintes (*PLC*) afin de produire la programmation logique par hiérarchie de contraintes (*PLHC*). *PLC* et *PLHC* sont paramétrées par D qui est le domaine des contraintes. De plus la *PLHC* est paramétrée par le comparateur C. Cette intégration de hiérarchies de contraintes dans la *PLC* permet à la théorie de la programmation logique d'être complétée par l'expressivité des contraintes de préférence.

Les travaux précédents concernant ce domaine sont scindés essentiellement en deux : la Programmation Logique par Contraintes (*PLC*) et l'utilisation des contraintes dans les applications. Le schéma de la *PLC* est décrit dans [JL87]. Un certain nombre d'interprètes *PLC* ont été implémentés Nons pouvons citer *Prolog III* [Col87], *CLP(R)* [HJM+87, JM87] et *CHIP* [DVS+88]. Il existe aussi des travaux considérables sur l'utilisation des contraintes dans les applications comme par exemple les contraintes géométriques, les simulations physiques, les interfaces utilisateurs, le dessin et le formatage de document, les algorithmes d'animations ou encore le dessin et l'analyse d'instruments mécaniques et de circuits électriques.

Dans cette section, nous illustrons l'extension de la notion de comparateur appliquée à la comparaison inter-hiérarchies. Les définitions précédentes des comparateurs données au chapitre 2 sont des cas particuliers dans le sens où elles utilisent une seule hiérarchie. Etant données les nouvelles définitions de cette section, il serait facile de voir que ces comparateurs inter-hiérarchies exhibent une conduite non monotone. On présentera aussi quelques propriétés intéressantes sur les hiérarchies de contraintes.

En programmation logique par contraintes, les règles sont de la forme : $p(t)$:- $q_1(t)$, ..., $q_m(t)$, $c_1(t)$, ..., $c_n(t)$ avec p, q_1, ..., q_m des symboles de prédicats et c_1, ..., c_n des contraintes. t dénote une liste de terme. En programmation logique par hiérarchie de contraintes les règles sont de la forme : $p(t)$:- $q_1(t)$, ..., $q_m(t)$, $s_1c_1(t)$, ..., $s_nc_n(t)$. s_i indique le niveau de préférence de la contrainte correspondante c_i. Des étiquettes sont associées aux différents niveaux de préférences de la hiérarchie.

L'utilisateur définit un nombre arbitraire d'étiquettes qui correspond au nombre de niveaux de la hiérarchie. Si toute étiquette s_i est "requise", alors il est clair que le programme est équivalent au même programme exprimé en *PLC* sans étiquettes sur les contraintes.

7.1.1 Fondement théorique de la comparaison inter-hiérarchies

Dans [Wil92, WB89, BDF+87], une solution d'un ensemble de hiérarchies de contraintes Δ, consiste en une valuation pour toutes les variables libres dans Δ. L'ensemble Δ consiste en des hiérarchies qui résultent des choix des règles alternatives dans un programme.

La définition de l'ensemble S de solutions au chapitre 2 reposait sur le fait que l'on considérait une seule hiérarchie. Ici, S contient toutes les solutions de Δ. Lorsque Δ contient une seule hiérarchie, la nouvelle définition de S est équivalente à celle du chapitre précédent. On définit en premier l'ensemble S_0 des valuations qui satisfont toutes les contraintes requises (par hiérarchie) des hiérarchies dans Δ. Chaque valuation θ dans S_0 est associée à la hiérarchie qu'elle satisfait. Ensuite l'ensemble S est défini comme avant, à l'exception du comparateur *meilleur* qui est paramétré par un ensemble de hiérarchies de contraintes. Par suite, les valuations potentielles d'une hiérarchie dans Δ qui sont jugées moins bonnes que d'autres valuations potentielles d'une autre hiérarchie dans Δ sont éliminées.

$$S_0 = \{\, \theta_h : \forall h \in \Delta\ \forall c \in h_0\ e\tau_0(c\theta) = 0 \}$$

$$S = \{\theta_h : \theta_h \in S_0 \wedge \forall \eta_{h'} \in S_0 \neg \mathit{meilleur}(\eta_{h'}, \theta_h, \Delta)\}$$

avec $\mathit{meilleur}(\eta_{h'}, \theta_h, \Delta) \Leftrightarrow (G(R(\Delta\eta_{h'})) <_G G(R(\Delta\theta_h)))$.

$S(C)$ est l'ensemble des solutions de l'ensemble des hiérarchies utilisant le comparateur C. Comme il est écrit au deuxième chapitre, les comparateurs sont irréflexifs et transitifs et respectent aussi la sémantique des hiérarchies dans Δ.

Puisque le comparateur *Localement-Meilleur* considère individuellement chaque contrainte de la hiérarchie pour déterminer l'effet d'une valuation, il serait inutile de considérer ce type de comparateur pour la comparaison inter-hiérarchies (c.à.d. pour comparer les différentes solutions des différentes hiérarchies dans Δ). En d'autres termes, *Localement-Meilleur* est défini seulement si Δ consiste en une seule hiérarchie.

Le schéma des comparateurs globaux *Globalement-Meilleur* a été étendu pour satisfaire la comparaison inter-hiérarchies. Nous rappelons que ce schéma est paramétré par une fonction g qui agrège les erreurs de toutes les contraintes d'un niveau de la hiérarchie. La définition de cette extension est :

Définition 7.1 :

la valuation θ_h est *Globalement-Meilleure* qu'une autre valuation $\eta_{h'}$ si pour chaque niveau jusqu'au niveau *k-1*, l'erreur obtenue par θ sur la hiérarchie *h* après agrégation est égale à celle obtenue par η sur la hiérarchie *h'* après agrégation, et au niveau *k* elle est strictement inférieure.

Pour une instance de $<_G$ dans *{Nombre-Contraintes-Non-Satisfaites, Somme-Pondérée-Prédicat, etc..}*, $V_{hi} = [v_1, v_2, ...,v_k]$ est la liste des erreurs obtenues du niveau *i* d'une hiérarchie $h \in \Delta$ en utilisant la fonction d'erreur triviale (qui retourne 0 si la contrainte est satisfaite et 1 sinon) et $g(V_{hi})$ est dans :

$$\left\{\left(\sum_{j=1}^{k} v_j\right)\left(\sum_{j=1}^{k} w_j v_j\right) \ldots\right\} \text{ avec } <_g \text{ est } < \text{ et } <>_g \text{ est } =.$$

Pour une instance de $<_G$ dans *{Somme-Pondérée, Pire-Cas, Moindre-Carré}*, $V_{hi} = [v_1, v_2, ...,v_k]$ est la liste des erreurs obtenues du niveau *i* d'une hiérarchie $h \in \Delta$ en utilisant la fonction d'erreur métrique et $g(V_{hi})$ est dans :

$$\left\{\left(\sum_{j=1}^{k} w_j v_j\right)\left(Max_{j:\,1..k}\{w_j v_j\}\right)\left(\sum_{j=1}^{k} w_j v_j^2\right)\right\} \text{ avec } <_g \text{ est } < \text{ et } <>_g \text{ est } = \text{ pour les réels.}$$

Pour une instance de $<_G$ appelée *globalement-meilleur*, pour deux vecteurs $V_{hi}=[v_1, v_2, ...,v_k]$ et $U_{hi'}=[u_1, u_2, ...,u_k]$, $<_g$ et $<>_g$ sont définis par :

$V_{hi} <_g U_{hi'} \equiv g(V_{hi}) <_g g(U_{hi'})$, et $V_{hi} <>_g U_{hi'} \equiv g(V_{hi}) <>_g g(U_{hi'})$.

Cette définition n'empêche pas l'ensemble Δ de hiérarchies d'être un ensemble infini. En pratique ceci peut avoir lieu lorsque les règles sont récursives et ne se terminent jamais comme le montre le programme suivant :

F(x) :- G(x), forte x<0.

G(1).

G(x):-G(x – 1).

Pour des programmes comme celui décrit ci-dessus, il n'y a pas de moyen d'éviter une recherche infinie pour trouver les meilleures solutions. Pour éviter de telles recherches, on doit s'assurer que le programme s'arrête bien.

Il est à signaler qu'une autre approche pour définir la comparaison inter-hiérarchies a été donnée par Michael Maher et Peter Stuckey [MS91]. Au lieu de définir l'ensemble S_0 des valuations qui sont solutions des contraintes requises des hiérarchies dans Δ, cette approche définit des pré-solutions pour chaque hiérarchie. Et une fonction de pré-mesure *g* qui associe une pré-solution et un ensemble de contraintes à un élement dans un ensemble *T*. Si cet ensemble est celui des réels et la fonction de pré-mesure est *0-k*, avec *k* le nombre de contraintes non satisfaites d'un niveau donné, alors cette approche génère le comparateur *Nombre-Contraintes-Non-Satisfaites*. Plusieurs autres comparateurs peuvent être obtenus en utilisant cette approche en faisant varier la fonction de pré-mesure et l'ensemble *T*.

7.1.2 Exemples en *PLHC* montrant le besoin de comparaison inter-hiérarchies

Les exemples que l'on décrit ici[1] ont un comportement inattendu bien que les solutions produites par l'interprète de la *PLHC* soient correctes au sens de la sémantique de la *PLHC*. Ceci résulte de l'incapacité de la *PLHC* à discriminer entre les solutions produites par les différents choix de règles. Dans les définitions initiales, les comparateurs sélectionnent la meilleure solution d'une seule hiérarchie. Ceci est appelé comparaison *intra-hiérarchie*. Ces exemples suggèrent le besoin d'une comparaison *inter-hiérarchies*.

FIGURE 40 : Programme montrant le besoin de la comparaison inter-hiérarchies

niveau([requise, forte])

F(x) :- G(x), forte x=3.

G(3);

G(4);

Si l'on considère le programme décrit dans la figure 40, et si on cherche à résoudre le but *F(A)* alors deux hiérarchies de contraintes résultent de ce programme. La première est : h_1 = *{requise A=3, forte A=3}* et la deuxième est : h_2 = *{requise A=4, forte A=3}*. Les deux hiérarchies sont des candidates potentielles pour résoudre le but *F(A)*.

On aurait souhaité n'avoir que la première hiérarchie qui produise la réponse *A=3* lors de sa résolution puisque cette réponse ne viole aucune contrainte dans cette hiérarchie. Pour réaliser ce souhait, la comparaison entre ces deux hiérarchies est indispensable. Cette comparaison n'aura un sens que si l'interprète de *PLHC* est paramétré par un comparateur global. Dans cet exemple, l'utilisation du comparateur global *Nombre-Contraintes-Non-Satisfaites* favorise la hiérarchie h_1 (puisque la réponse produite par cette hiérarchie ne viole aucune de ces contraintes) et élimine la hiérarchie indésirable h_2 (puisque la réponse produite par cette hiérarchie qui est *A=4* viole la contrainte *forte A=3*). Cette comparaison n'aura pas de sens si l'interprète de *PLHC* est paramétré par le comparateur *Localement-Prédicat-Meilleur* puisque ce comparateur considère les contraintes individuellement et ne peut pas comparer deux solutions qui satisfont deux ensembles de contraintes non inclus l'un dans l'autre.

D'une manière générale, puisque le comparateur *Localement-Meilleur* considère individuellement chaque contrainte de la hiérarchie pour déterminer l'effet d'une valuation, il serait inutile de considérer ce type de comparateur pour la comparaison inter-hiérarchies (c.à.d. pour comparer les différentes solutions des différentes hiérarchies).

Il y a plusieurs autres exemples où la comparaison inter-hiérarchies conduit à la suppression de solutions indésirables de l'ensemble des réponses. Le deuxième exemple décrit dans la figure 41 consiste en un programme qui compte le nombre minimal de pièces de monnaie échangées contre une somme spécifique donnée. Sans comparaison inter-hiérarchies, la solution au but *echange(16, Pièce-20, Pièce-10, Pièce-5, Pièce-1)* va essentiellement ignorer les contraintes de préférences, puisque les valeurs des variables *Pièce-20, Pièce-10, Pièce-5, Pièce-1* doivent être déterminées par le prédicat *Int*. La solution souhaitée : *Pièce-20 = 0, Pièce-10 = 1, Pièce-5 = 1* et *Pièce-1=1* est retournée par l'interprète paramétré par le comparateur *Localement-Prédicat-Meilleur*, mais aussi plusieurs autres solutions sont retournées notamment *Pièce-1=16*. Tandis qu'avec une comparaison inter-hiérarchies en utilisant le comparateur global *Nombre-Contraintes-Non-Satisfaites*, seule la solution souhaitée est retournée.

1 Ces exemples sont empruntés de [WB89,Wil92,DVS+88, MS91]

FIGURE 41 : Programme monnaies-échangées

```
niveaux ([requise, forte])
echange(Somme, Pièce-20, Pièce-10, Pièce-5, Pièce-1) :-
    requise (Pièce-20 × 20) + (Pièce-10 × 10) +(Pièce-5 × 5) + (Pièce-1 × 1 ) = Somme,
    disponible(Dp-20, Dp-10, Dp-5, Dp-1),
    requise Pièce-20 ≤ Dp-20, requise Pièce-10 ≤ Dp-10,
    requise Pièce-5 ≤ dp-5, requise Pièce-1 ≤ Dp-1,
    int(Pièce-20), int(Pièce-10), int(Pièce-5), int(Pièce-1),
    forte Pièce-1 < 5, forte Pièce-5 < 2, forte Pièce-10 < 2.
int(0).
int(X) :- requise X>0, int(X−1).
disponible (10, 2, 4, 16).
```

7.1.3 L'aspect non-monotone des comparateurs

La logique classique est monotone dans le sens où ajouter un nouvel axiome à la théorie ne peut qu'agrandir l'ensemble des théorèmes. Les comparateurs globaux définis auparavant soulèvent la non-monotonie des programmes puisque l'ajout d'une nouvelle règle peut créer une incohérence avec les solutions précédentes. Un comparateur est monotone si l'ensemble de solutions de Δ est un sous ensemble de celui de $\Delta \cup \Delta'$ avec Δ' un ensemble quelconque de hiérarchies de contraintes.

Définition 7.2[1] :

Soit Δ et Δ' deux ensembles de hiérarchies de contraintes. Soit θ une valuation et C un comparateur global. C est monotone si et seulement si :

$\forall\Delta\ \forall\Delta'\ \forall\theta$ si $\theta_h \in S_\Delta(C)$ alors $\theta_h \in S_{\Delta\cup\Delta'}(C)$.

Proposition 7.1 : Soit D un domaine contenant plus qu'un élément. Chaque comparateur global respectant la hiérarchie est non-monotone.

Preuve : Soit Δ=*{{requise x=a, défaut x=b}}* et soit Δ'=*{{requise x=b}}* avec a et b sont deux éléments distincts du domaine D. Soit C un comparateur qui respecte l'ensemble des hiérarchies. $S_\Delta(C)$=*{{a}}* et $S_{\Delta'}(C)$=*{{b}}*. La valuation dans $S_{\Delta'}(C)$ est meilleure que celle dans $S_\Delta(C)$, ainsi la valuation $x=a$ n'est pas dans $S_{\Delta\cup\Delta'}(C)$ ($S_{\Delta\cup\Delta'}$ = *{{b}}*). On conclut donc que C est non-monotone.

Pour illustrer la propriété de la non-monotonie et ses effets sur un programme, considérons le programme suivant :

F(x) :- G(x), forte x>0.

G(−1).

Etant donné le but *F(A)*, un interprète pour *PLHC(R,C)* retournera la réponse *A=−1*.

Supposons maintenant que l'on ajoute le fait *G(1)* au programme. Si l'interprète est basé sur le comparateur local *Localement-Meilleur*, les réponses retournées seront *A=−1* et *A=1* puisque ce type de

1. Cette définition ne s'applique pas à des comparateurs locaux puisque l'ensemble $S_{\Delta\cup\Delta'}$ est non défini.

comparateur ne permet pas de réaliser une comparaison inter-hiérarchies. Cependant, l'utilisation de n'importe quel comparateur global *C* qui respecte l'ensemble des hiérarchies dans un interprète pour *PLHC(R,C)* retourne la seule réponse *A=1* à la question *F(A)*. Ainsi, l'ensemble des faits déduits du programme initial n'est pas un sous-ensemble de l'ensemble des faits dérivés lors de l'ajout d'une nouvelle règle. Ceci est similaire à la logique non-monotone dans laquelle l'ajout d'un axiome à l'ensemble d'axiomes existant peut résulter en une falsification des théorèmes existants.

7.1.4 Contraintes non primitives

Les interprètes existants de *PLHC* permettent seulement l'expression des contraintes primitives (c.à.d. contraintes de la forme $c(t_1, \ldots, t_n)$). Cependant, le résultat formel dans [BMM+89] est valide si l'on permet une large classe de contraintes formée par la combinaison de contraintes primitives utilisant les connecteurs logiques (*et*, *ou* et *non*) et les quantificateurs existentiels. En *PLC*, les effets de la disjonction et de la conjonction peuvent être réalisés indirectement. Tandis qu'en *PLHC*, ceci n'est pas vrai comme le montre l'exemple dans cette section.

Si l'on essayait d'étendre le formalisme *PLHC* de façon à inclure des contraintes non primitives, alors cette extension ajouterait une grande expressivité et puissance à ces contraintes et par conséquent on attribuerait d'autres sens aux programmes qui seraient différents des sens initiaux. De plus, il y a plusieurs exemples où l'utilisation de la disjonction élimine la nécessité de comparaisons inter-hiérarchies. Il est bien évident que tous les cas ne peuvent pas être traités de cette façon.

En *PLC*, la conjonction de contraintes est accompli par l'ajout de ces contraintes dans le corps d'une règle et la disjonction (au moins sémantiquement) par l'ajout d'une nouvelle règle au programme. En *PLHC*, on ne peut pas atteindre ce résultat si simplement. Considérons la hiérarchie suivante : *H={requise x >=−10, requise x ≤ 10, défaut x >= 5 , défaut x≤−5}*. En utilisant le comparateur *Localement-Prédicat-Meilleur*. L'ensemble des solutions de cette hiérarchie est *{x / −10 ≤ x ≤−5* ou *5 ≤ x ≤ 10}*. Supposons maintenant que l'on reformule cette hiérarchie en mettant une conjonction entre les contraintes étiquetées par *défaut* (c.à.d on remplace dans la hiérarchie *défaut x >= 5 , défaut x ≤−5* par *défaut (x >= 5∧ x ≤−5)*). L'ensemble des solutions à cette nouvelle hiérarchie en utilisant le même comparateur est *{x / −10 ≤ x ≤10}*. On voit bien que l'on ne peut pas atteindre les effets de la conjonction en *PLHC* seulement par l'ajout de contraintes au corps des règles comme on fait en *PLC*.

Pareillement, la disjonction ne peut pas être atteinte en *PLHC* par un simple ajout de règles représentant des directions alternatives pour la satisfaction des contraintes. Considérons l'exemple suivant :

F(x) :- G(x), forte x>0.

G(−1).

G(1).

Les réponses au but *F(A)* en utilisant le comparateur *Localement-Prédicat-Meilleur* sont *A=−1* et *A=1*. Supposons que l'on étende le formalisme *PLHC* pour inclure non seulement les contraintes primitives dans le corps d'une règle mais aussi la disjonction des contraintes primitives. Par exemple comme dans le programme suivant :

F(x) :- requise (x=−1 ∨ x=1), forte x>0.

La réponse au but *F(A) est A=1*. Ceci soulève deux possibilités intéressantes selon [Wil92]. La première est que la portée des programmes possibles en *PLHC* doit être étendue par l'extension des classes des contraintes pour inclure les connecteurs de conjonction et de disjonction. La deuxième est de réaliser les effets de la comparaison inter-hiérarchies en permettant le disjonction de contraintes primitives comme le montre l'exemple précédent. Ceci accomplit en effet le mélange de deux hiérarchies en une seule. Cependant, il n'est pas toujours possible de combiner deux hiérarchies de cette façon, comme le montre l'exemple suivant :

F(x) :- requise x=1, forte x≤ 0.

F(x) :- requise x=2 .

En satisfaisant le but *F(A)*, deux hiérarchies sont créées, la première consiste en une contrainte requise et une contrainte forte et la deuxième consiste en une contrainte requise. Le comparateur *localement-prédicat-meilleur* retourne les deux solutions *A=1* et *A=2*. Un comparateur global qui respecte la hiérarchie retournera la solution *A=2*. Si l'on essaye de réécrire ce programme en utilisant la disjonction pour que le comparateur *Localement-Prédicat-Meilleur* retourne la solution *A=2*, on doit déterminer une technique pour combiner les contraintes requises. La seule possibilité est le programme suivant :

F(x) :- requise (x=1 ∨ x=2) , forte x≤ 0.

Malheureusement, les solutions au but *F(A)* obtenues par le comparateur *localement-prédicat-meilleur* sont les mêmes qu'avant à savoir *A=1* et *A=2*. Cette transformation est pire dans la mesure où certains comparateurs globaux (par exemple *Somme-Pondérée-Métrique*) retournent la réponse fausse *A=1* (puisque cette réponce satisfait mieux la contrainte *forte x ≤ 0*). Ceci est dû au fait que l'on a mis en relation une contrainte requise d'une hiérarchie avec une contrainte de préférence d'une autre hiérarchie ce qui entraîne la perte de l'intention (c.à.d correction) du programme initial.

Le programme de la figure 42 est un exemple plus réaliste dans lequel la disjonction n'est pas un remède contre le besoin d'avoir une comparaison inter-hiérarchies. Dans ce programme, on essaye de trouver un temps convenable pour une réunion entre Bertrand et Franck. Bertrand est libre Vendredi de 12h à 16h, mais il préfère que la réunion soit après 14h. Il est aussi libre Samedi de 10h à 13h. Franck est libre Vendredi de 13h à 14h et Samedi de 11h à 12h. Même si Bertrand est disponible ces deux jours aux mêmes horaires, on aimerait que la réponse à la question *date-réunion-possible(Bertrand, Franck, Jour, Heure, 1)* retourne une réponse n'incluant pas "Vendredi à 13h" puisque cette solution ne satisfait pas la contrainte de préférence posée par Bertrand. Cependant, la réponse "Samedi à 11h" doit satisfaire toutes les contraintes dans la hiérarchie qui l'a produite et doit être la seule.

La résolution du but *Date-réunion-possible(Bertrand, Franck, Jour, Heure, 1)* entraîne deux hiérarchies de contraintes différentes. La première consiste uniquement en des contraintes requises et la deuxième en des contraintes requises (différentes des précédentes) et la contrainte de préférence *forte Début-libre-2 >= 14.* Il n'y a pas de possibilité de reformuler ce programme avec une seule hiérarchie utilisant la disjonction puisqu'on ne peut pas permettre à la contrainte de préférence de la seconde hiérarchie d'éliminer des valuations qui résolvent les contraintes requises de la première hiérarchie. Bien qu'on étende le formalisme *PLHC* en incluant les contraintes non-primitives pour pouvoir réécrire quelques programmes et éliminer les solutions indésirables, on ne peut pas utiliser cette technique pour se dispenser du besoin de comparaison inter-hiérarchies.

FIGURE 42 : Programme nécisitant une comparaison inter-hierarchies

```
niveaux ([requise, forte])
date-réunion-possible(Personne-1, Personne-2, Jour, Heure, Durée) :-
    libre(Personne-1, Jour, Début-libre-1, Fin-libre-1),
    libre(Personne-2, Jour, Début-libre-2, Fin-libre-2),
    requise Début-libre-1 ≤ Heure,
    requise Début-libre-1 ≤ Heure,
    requise Heure + Durée ≤ Fin-libre-1,
    requise Heure + Durée ≤ Fin-libre-2.
libre(Franck, Samedi, 11, 12).
libre(Franck, Vendredi, 13, 14).
libre(Bertrand, Samedi, 10, 13).
libre(Bertrand, Vendredi, T, 16) :- requise T>= 12, forte T >= 14.
```

7.2 Vue générale de l'algorithme proposé

D'une manière opérationnelle, les buts sont exécutés comme en *PLC* en ignorant temporairement les contraintes de préférences et en les accumulant dans des hiérarchies que l'on met dans l'ensemble Δ (cet ensemble contiendra les hiérarchies résultant des différents choix des règles alternatives dans le programme). L'objectif de l'algorithme que nous allons présenter est l'élimination de certaines hiérarchies de l'ensemble Δ. La résolution de ces hiérarchies produiraient des solutions qui seraient jugées par le critère de comparaison utilisé dans le programme (par exemple un des trois critères intégrés dans Houria) moins bonnes que d'autres solutions produites par d'autres hiérarchies dans Δ. En général, ces solutions jugées moins bonnes sont non intuitives et indésirables. Pour atteindre cet objectif, on décrit trois phases en utilisant deux techniques d'optimisation.

La première phase consiste en la réalisation des trois opérations suivantes :

1. Pour chacune des hiérarchies dans Δ, on calcule par l'opérateur $\oplus_{Comp}$ la somme des paires *(étiquette, poids)* des contraintes n'ayant que des variables liées (toutes les variables de ces contraintes sont affectées à l'issue de la résolution des prédicats du programme). Cette somme est dénotée par *ep-liée*.

2. Pour chacune des hiérarchies dans Δ, on calcule par l'opérateur $\oplus_{Comp}$ la somme des paires *(étiquette, poids)* des contraintes possédant au moins une variable libre. Cette somme est dénotée par *ep-libre*.

3. On élimine de l'ensemble Δ, les hiérarchies ayant la somme de leurs *ep-liée* et *ep-libre* inférieure ou égale[1] à la *ep-liée* d'une autre hiérarchie dans Δ (puisque on est certain que cette dernière produirait une solution qui serait jugée meilleure par le critère utilisé).

1. Si l'on désire avoir toutes les solutions, on doit remplacer inférieure ou égale par inférieure strictement.

La deuxième phase consiste à la réalisation des deux opérations suivantes :

1. On ordonne les hiérarchies restantes dans Δ selon l'ordre décroissant sur le critère : somme des *ep-liée* et *ep-libre* de chacune des hiérarchies.

2. Les hiérarchies dans Δ sont traitées l'une après l'autre en faisant appel au résolveur Houria. Houria extrait d'une hiérarchie *h* possédant au moins une variable libre un sous-ensemble maximal consistant (voir chapitre 5) de contraintes (dénoté par Gr_h) qui peut être résolu. Cette opération se termine si un des deux cas suivants se produit :

- toutes les hiérarchies dans Δ ont été traitées,

- la somme de *ep-liée* de *h* et des paires *(étiquette, poids)* dans Gr_h est supérieure ou égale[1] à celle des *ep-liée* et *ep-libre* de la hiérarchie qui suit *h* dans Δ. Dans ce cas, toutes les hiérarchies qui suivent *h* sont enlevées de Δ. Il est inutile d'appeler Houria sur ces hiérarchies enlevées, puisque Δ est ordonné et par conséquent on est certain que les solutions à ces hiérarchies seront jugées par le critère utilisé dans le programme moins bonnes que la solution à la hiérarchie *h*.

La troisième et dernière phase consiste en la réalisation des deux opérations suivantes :

1. On élimine de l'ensemble des hiérarchies restantes dans Δ les hiérarchies ayant chacune la somme de leur liste *ep-liées* et des paires *(étiquette, poids)* dans leur graphe *Gr* de méthodes consistantes inférieure strictement à celle d'une autre hiérarchie dans Δ .

2. Pour chacune des hiérarchies restantes dans Δ, on résout l'ensemble des contraintes dans son graphe *Gr* de méthodes consistantes associé. Cette résolution consiste à déterminer les valeurs des variables libres dans ces contraintes.

7.3 Mise en oeuvre de l'algorithme proposé

Dans cette section, on présente les deux procédures *Initialisation* et *Solutions-Désirées,* qui mettent en oeuvre l'approche algorithmique décrite précédemment pour la réalisation de la comparaison inter-hiérarchies.

Le premier pas de la procédure *Initialisation* consiste à former l'ensemble Δ des hiérarchies résultant des différents choix de règles des prédicats dans le programme (ligne 1; figure 43) et de la résolution des contraintes requises. On supose que cet ensemble est fini (le programme est bien écrit). Dans le cas où l'ensemble Δ des hiérarchies est infini, un tel pas ne retourne pas de solution trouvée car il ne peut pas échapper aux branches infinies dans l'arbre de recherche.

Successivement, la procédure examine l'ensemble Δ et associe à chacune des hiérarchies de cet ensemble :

la liste *V-libre* contenant l'ensemble des variables libres de cette hiérarchie (l'ensemble des variables qui n'ont pas été déterminées par le premier pas de la procédure) (ligne 3; figure 43),

la liste *C* contenant l'ensemble des contraintes de cette hiérarchie (ligne 4; figure 43),

la liste *C-éligible* contenant l'ensemble des contraintes de cette hiérarchie ayant chacune au moins une variable libre, cette liste sera utilisée pour affecter les variables de la liste *V-libre* (ligne 5; figure 43),

1. On peut remplacer supérieure ou égale par supérieure stricte si l'on désire avoir toutes les solutions désirables.

la liste *C-tester* contenant l'ensemble des contraintes de cette hiérarchie ne possédant aucune variable dans la liste *V-libre* (ligne 6; figure 43). Cette liste va contribuer à l'élimination des hiérarchies telles que leurs résolutions produiraient des solutions indésirables

la variable *ep-liée* contenant la somme des paires *(étiquette, poids)* des contraintes satisfaites dans la liste *C-tester* de cette hiérarchie (lignes 7, 8, 9, 10; figure 43). La notion de la somme dans ce chapitre est relative au critère utilisé. Cette somme est obtenue par l'opérateur $\oplus_{Comp.}$

On examine les contraintes éligibles de chaque hiérarchie dans Δ. Pour chacune de ces contraintes, la procédure marque les variables liées (n'appartenant pas à la liste *V-libre*) par l'annotation *lecture-seulement* (voir paragraphe 1.1 du chapitre 3). Ceci pour ne pas redéterminer ces variables, du fait qu'elles ont été calculées par des contraintes requises (c.à.d. prédicats dans le programme) dans le premier pas de la procédure (lignes 11-15; figure 43).

FIGURE 43 : Procédure d'initialisation de l'ensemble Δ

Procédure Initialisation(P, Comp)

1 Δ ← l'ensemble des hiérarchies résultant des choix de règles dans le programme P.
2 Pour chaque hiérarchie h ∈ Δ faire:
Début
3 $V\text{-libre}_h$ ← l'ensemble des variables libre dans h, V_h ← les variables dans h.
4 C_h ← l'ensemble des contraintes dans h.
5 $C\text{-éligible}_h$ ← contraintes ∈ C_h ayant chacune au moins une variable dans $V\text{-libre}_h$
6 $C\text{-tester}_h$ ← contraintes ∈ C_h n'ayant aucune variable dans $V\text{-libre}_h$.
7 $ep\text{-liée}_h$ ← ∅.
8 Pour chaque c ∈ $C\text{-tester}_h$ faire:
9 Si c est satisfaite alors $ep\text{-liée}_h$ ← ($ep\text{-liée}_h$ $\oplus_{Comp}$ $\bar{c}$) Fin-Si.
10 Fin-Pour.
11 Pour chaque c ∈ $C\text{-éligible}_h$ faire:
12 Pour chaque variable v contrainte par c faire:
13 Si v ∉ $V\text{-libre}_h$ alors marquer v par l'annotation lecture-seulement.
14 Fin-Pour.
15 Fin-Pour.
Fin.
16 Fin-pour.

La procédure *Solutions-Désirées* met en œuvre les trois phases décrites précédemment afin de ne garder dans Δ que les hiérarchies telles que leur résolution produiraient les solutions souhaitables.

La procédure élimine de l'ensemble Δ les hiérarchies qui ont la somme des paires *(étiquette, poids)* des contraintes dans la liste *C-éligible* et de la liste *ep-liée* de cette hiérarchie inférieure strictement à la liste *ep-liée* d'une autre hiérarchie dans Δ (ligne 1; figure 44).

L'ensemble Δ résultant est ordonné. Cet ordonnancement aidera par la suite à prédire s'il est nécessaire de traiter toutes les hiérarchies dans Δ (ligne 2; figure 44). La procédure enlève la hiérarchie située en tête de Δ (ligne 5; figure 44) et appelle le résolveur Houria sur la liste *C-éligible* avec le comparateur utilisé dans le programme (ligne 6; figure 44). Le résultat de cet appel est mis dans le couple (*Gr, ep-Gr*) qui contient (l'ensemble maximal consistant de contraintes extrait de la liste *C-éligible*, la somme des paires *(étiquette, poids)* des contraintes de ce graphe solution) (ligne 6; figure 33). La hiérarchie ainsi traitée est rangée dans l'ensemble temporaire Δ'. Ces opérations sont exécutées jusqu'à ce que l'ensemble Δ soit vide ou jusqu'à ce qu'on soit certain qu'il est inutile de traiter les hiérarchies restantes dans Δ (du fait que Δ est ordonné et que par conséquent leurs résolutions ne produiraient pas une solution souhaitable) (ligne 8; figure 44).

La procédure choisit un sous-ensemble de hiérarchies dans Δ' et l'affecte à Δ (ligne 9; figure 44). Chacune de ces hiérarchies choisies possède la somme de ces listes respectives *ep-Gr* et *ep-liée* maximale (c.à.d toutes les hiérarchies dans Δ possèdent la même somme, de plus cette somme est strictement supérieure à chacune des hiérarchies non choisies restantes dans Δ'). Pour chacune des hiérarchies dans Δ (on peut traiter une seul hiérarchie si on souhaite n'avoir qu'une reponse) la procédure appelle la procédure de résolution pour résoudre le *Gr* associé à cette hiérarchie (lignes 9, 10; figure 44). Cette résolution consiste à déterminer les valeurs des variables dans la liste *V-libre* en exécutant les méthodes des contraintes dans *Gr* et en appliquant l'algorithme de propagation locale. Après cette opération, toutes les variables de l'ensemble *V* d'une hiérarchie dans Δ sont maintenant déterminées et constituent une solution (ou réponse) désirée du programme *P* (lignes 12; figure 44).

FIGURE 44 : Procédure de construction des solutions souhaitables

Procédure Solutions-Désirées(Comp,D)

1 $\Delta \leftarrow \Delta - \{h \mid \exists h' \in \Delta \wedge ep\text{-}liée_{h'} >_{Lex} ep\text{-}liée_h \oplus_{Comp} (\{ \bar{c} \mid c \in C\text{-}éligible_h\})\}$.

2 ordonner l'ensemble Δ *par le critère (ep-liée* $\oplus_{Comp}$ $(\bar{c} \mid c \in C\text{-}éligible))$ *décroissant.*

3 Soient la variable booléenne fini initialisée à faux et l'ensemble Δ' *initialisé à* ∅.

4 Tant que $\neg$ *(fini* $\vee$ Δ = ∅*) faire:*

Début

5 $\Delta \leftarrow \Delta - h$.

6 $(Gr_h , ep\text{-}Gr_h) \leftarrow Houria(Comp, C\text{-}éligible_h)$.

7 $\Delta' \leftarrow \Delta' + h$.

8 *Si* $(ep\text{-}Gr_h \oplus_{Comp} ep\text{-}liée_h) \geq_{Lex} (ep\text{-}liée_{(Tete(\Delta))} \oplus_{Comp} (\{ \bar{c} \mid c \in C\text{-}éligible_{(Tete(\Delta))}\}))$ *alors fini* $\leftarrow$ *vrai Fin-Si.*

Fin

Fin-Tantque.

9 $\Delta \leftarrow \{h \in \Delta' / \forall h' \in \Delta' \neg((ep\text{-}Gr_{h'} \oplus_{Comp} ep\text{-}liée_{h'}) >_{Lex} (ep\text{-}Gr_h \oplus_{Comp} ep\text{-}liée_h))\}$.

10 Pour chaque $h \in \Delta$ *faire:*

Début

11 *résoudre* Gr_h *en utilisant l'algorithme de la propagation locale dans Houria.*

12 *retourner*(V_h).

Fin.

Fin-Pour.

7.4 Exemple illustratif en PLHC

On présente un exemple de programmation logique par hiérarchie de contraintes pour illustrer le déroulement des deux procédures décrites dans la section précédente.

On utilise dans cet exemple le comparateur global *Nombre-Contraintes-Non-Satisfaites* (pour simplifier les notations, on considère que les contraintes sont uniquement étiquetées puisque ce comparateur ne manipule pas de poids) .

FIGURE 45 : Exemple de comparaison inter-hiérarchies utilisant le comparateur τ_{NCNS}

niveaux ([requise, forte, moyenne, faible, très-faible]).

Comparateur ([Comp = Nombre-Contraintes-Non-Satisfaites]).

F(x,y,z) :- G(x), forte x=2y+z , moyenne x=2y, faible x=4.

F(x,y,z) :- G(x), forte x=8, moyenne x=2y+z, faible x=2y.

F(x,y,z) :- K(x,y), forte x=2y, moyenne x=2y+z, moyenne y=2x+z, très-faible x=4.

G(4).

G(8).

K(1,3).

K(6,3).

Etant donné le but *F(A,B,C)*, le premier pas de la procédure *Initialisation(P,Comp)* retourne l'ensemble Δ contenant six hiérarchies ($h_1..h_6$) résultant des choix alternatifs des règles des prédicats *G* et *K*.

h_1 *={ requise A=4, forte A=2B+C, moyenne A=2B, faible A=4}*

h_2 *={ requise A=8, forte A=2B+C, moyenne A=2B, faible A=4}*

h_3*={ requise A=4, forte A=8, moyenne A=2B+C, faible A=2B}*

h_4*={ requise A=8, forte A=8, moyenne A=2B+C, faible A=2B}*

h_5*={ requise A=1, requise B=3, forte A=2B, moyenne A=2B+C, moyenne B=2A+C, très-faible A=4}*

h_6*={ requise A=6, requise B=3, forte A=2B, moyenne A=2B+C, moyenne B=2A+C, très-faible A=4}.*

Le résultat de la procédure *Initialisation* sur les six hiérarchies dans Δ est :

($V\text{-}libre_{h1}$, $C\text{-}éligible_{h1}$, $C\text{-}tester_{h1}$, $ep\text{-}liée_{h1}$) *=({B,C}, {forte A=2B+C, moyenne A=2B}, {faible A=4}, {faible}).*

($V\text{-}libre_{h2}$, $C\text{-}éligible_{h2}$, $C\text{-}tester_{h2}$, $ep\text{-}liée_{h2}$) *=({B,C}, {forte A=2B+C, moyenne A=2B}, {faible A=4}, {}).*

($V\text{-}libre_{h3}$, $C\text{-}éligible_{h3}$, $C\text{-}tester_{h3}$, $ep\text{-}liée_{h3}$) *=({B,C},{moyenne A=2B+C, faible A=2B}, {forte A=8}, {}).*

($V\text{-}libre_{h4}$, $C\text{-}éligible_{h4}$, $C\text{-}tester_{h4}$, $ep\text{-}liée_{h4}$) *=({B,C},{moyenne A=2B+C, faible A=2B}, {forte A=8}, {forte}).*

($V\text{-}libre_{h5}$, $C\text{-}éligible_{h5}$, $C\text{-}tester_{h5}$, $ep\text{-}liée_{h5}$) *=({C},{moyenne A=2B+C, moyenne B=2A +C} , { forte A=2B, très-faible A=4}, {}).*

($V\text{-}libre_{h6}$, $C\text{-}éligible_{h6}$, $C\text{-}tester_{h6}$, $ep\text{-}liée_{h6}$) *=({C},{moyenne A=2B+C, moyenne B=2A+C} , { forte A=2B, très-faible A=4}, {forte}).*

Le premier pas de la procédure *Solutions-Désirées* élimine de Δ les hiérarchies h_5 et h_3 puisqu'on a respectivement :

$ep\text{-}liée_{h6} >_{lex} ((\oplus_{NCNS}\ C\text{-}éligible_{h5}) \oplus_{NCNS} ep\text{-}liée_{h5})$ *(c.à.d. (forte)* $>_{lex}$ *(moyenne, moyenne)),*

et $ep\text{-}liée_{h6} >_{lex} ((\oplus_{NCNS}\ C\text{-}éligible_{h3}) \oplus_{NCNS} ep\text{-}liée_{h3})$ *(c.à.d. (forte)* $>_{lex}$ *(moyenne, faible)).*

Après ce premier pas de la procédure, l'ensemble Δ est ordonné selon le critère $((\oplus_{NCNS}\ C\text{-}éligible_h) \oplus_{NCNS} ep\text{-}liée_h)$ et contient : $\{h_6, h_4, h_1, h_2\}$.

La procédure appelle Houria pour traiter les contraintes éligibles des hiérarchies dans l'ensemble Δ. La hiérarchie h_6 est enlevée de Δ pour traitement. L'appel de Houria sur cette hiérarchie retourne le couple $(Gr_{h6}, ep\text{-}Gr_{h6})$. Le graphe extrait Gr_{h6} des contraintes éligibles de la hiérarchie h_6 contient la seule méthode $C \leftarrow A\text{-}2B$ (c.à.d la valeur de la variable C peut être déterminée à partir de celle de A et de B) (les autres méthodes de la contrainte $C = A\text{-}2B$ ne sont pas utilisées ici puisque les variables A et B sont marquées par la notation *lecture-seulement* dans cette hiérarchie). Le contenu de la liste $ep\text{-}Gr_{h6}$ est : *(moyenne)*. Un autre graphe alternatif contenant la méthode $C \leftarrow B\text{-}2A$ est calculé par Houria, mais puisque ce graphe possède la même liste d'étiquettes que celle du graphe Gr_{h6}, on ne donne qu'une seule solution.

Après le traitement de la hiérarchie h_6, aucune des deux conditions d'arrêt n'est réalisée (c.à.d $\neg(fini \vee \Delta = \varnothing)$) et par conséquent, on est pas certain d'avoir obtenu la solution optimale du programme P.

La hiérarchie h_4 est enlevée de Δ pour traitement. La procédure appelle Houria pour traiter les contraintes éligibles de cette hiérarchie. Le résultat retourné à la suite de cet appel est le couple $(Gr_{h4}, ep\text{-}Gr_{h4})$. Le graphe Gr_{h4} des méthodes extrait des contraintes éligibles de la hiérarchie h_4 contient $\{B \leftarrow A/2,\ C \leftarrow A\text{-}2B\}$ et $ep\text{-}Gr_{h4}$ contient : *(moyenne, faible)*.

Après cette opération, la hiérarchie h_1 n'est pas traitée[1] puisque l'ensemble Δ est ordonné et que même si toutes les contraintes éligibles dans h_1 étaient consistantes entre elles, sa résolution ne produirait pas une solution meilleure que celle qui sera produite par la hiérarchie h_4 :

$((\oplus_{NCNS}\ C\text{-}éligible_{h1}) \oplus_{NCNS} ep\text{-}liée_{h1})) \leq_{Lex} (ep\text{-}Gr_{h4} \oplus_{NCNS} ep\text{-}liée_{h4})$

(c.à.d. (forte, moyenne, faible) $\leq_{Lex}$ *(forte, moyenne, faible)).*

L'ensemble Δ' après ces différents pas contient les deux hiérarchies h_4 et h_6. Seule la hiérarchie h_4 est affectée à Δ puisque la hiérarchie h_6 est éliminée du fait que :

$(ep\text{-}Gr_{h6} \oplus_{NCNS} ep\text{-}liée_{h6}) \leq_{Lex} (ep\text{-}Gr_{h4} \oplus_{NCNS} ep\text{-}liée_{h4})$

(c.à.d (forte, moyenne, très-faible) $\leq_{Lex}$*(forte, moyenne, faible)).*

La procédure appelle Houria pour la résolution des méthodes dans Gr_{h4} et la solution désirée $A=8$, $B=4$, $C=0$ est retournée.

1. En ce point, on est certain qu'au moins une des hiérarchies qui donne la solution optimale lors de sa résolution a été traitée.

Synthèse du chapitre

L'objet de ce chapitre a été de montrer en partie l'intérêt de l'utilisation de comparateurs globaux dans un résolveur tel que celui conçu aux chapitres 5 et 6 pour réaliser la comparaison inter-hiérarchies dans le cadre de la *PLHC*. Cette comparaison inter-hiérarchies est utile dans plusieurs applications contenant des hiérarchies de contraintes fonctionnelles, comme dans les contraintes géométriques, les simulations physiques, les interfaces graphiques, le formatage des documents, les algorithmes d'animation, ou encore l'analyse et le dessin de circuits électriques et d'appareils mécaniques.

L'extension théorique de l'ensemble de solutions S qui considère cette comparaison inter-hiérarchies décrite dans [WB89] et le résolveur Houria, nous ont permis de concevoir l'algorithme décrit dans ce chapitre. Cet algorithme prédit l'ensemble des hiérarchies qui vont produire les solutions optimales d'un programme donné. On intègre également dans cet algorithme des techniques d'optimisation qui gardent la complétude et qui reposent sur l'importance accordée aux contraintes.

L'algorithme présenté dans ce chapitre peut être incorporé dans les langages de programmation logique par hiérarchies de contraintes. Ceci permettra d'exécuter la comparaison inter-hiérarchies et donc de pouvoir éliminer les hiérarchies telles que leurs résolutions produiraient des solutions non désirables (c.à.d. on n'a pas besoin de résoudre toutes les hiérarchies et ensuite comparer leurs solutions pour éliminer les moins bonnes. Ici on détecte le sous ensemble des hiérarchies qui va produire la meilleure valuation selon le critère utilisé).

Cet algorithme est plus prometteur qu'un algorithme du type séparation et évaluation lorsque le nombre de choix de règles alternatives d'un prédicat du programme est petit car :

- si la solution est du côté droit de l'arbre[1],

  le temps de trouver cette solution en utilisant un algorithme du type séparation et évaluation est nettement supérieur à celui mis par notre approche puisque l'algorithme du type séparation et évaluation doit balayer tout l'arbre pour arriver à cette solution (c.à.d à chaque construction d'une branche prometteuse l'appel à Houria est nécessaire pour déterminer l'importance de la branche), tandis qu'avec notre approche, l'ensemble Δ est trié et donc on détecte rapidement la hiérarchie qui produira la solution(c.à.d. le nombre d'appel à Houria est proche de l'optimal).

- si la solution est du côté gauche de l'arbre,

  notre approche est équivalente à un algorithme du type séparation et évaluation en terme du nombre d'appels à Houria. Cependant, l'algorithme du type séparation et évaluation doit balayer le reste de l'arbre (sans faire appel à Houria) pour s'assurer qu'il a déjà obtenu la solution.

- d'une manière générale :

  plus la solution est du côté droit de l'arbre, plus on a intérêt à utiliser notre approche.

  plus la solution est du côté gauche de l'arbre, plus notre approche converge vers un algorithme du type séparation et évaluation.

1. On fait référence à l'arbre de résolution dessiné à plat devant le lecteur.

Lorsque le nombre de choix de règles alternatives d'un prédicat du programme est grand et qu'en moyenne, chaque règle utilise un ou plusieurs prédicats dans sa queue alors il s'agit d'un problème combinatoire et par conséquent le nombre de hiérarchies est exponentiel (problème de mémoire et de temps). L'utilisation de notre approche dans ce cas n'est donc pas raisonnable. L'idée ici serait d'utiliser un algorithme du type séparation et évaluation qui ferait appel à Houria.

Le critère d'évaluation dépendra du critère utilisé dans le programme. Par exemple si le critère utilisé est le *Nombre-Contraintes-Non-Satisfaites* alors le critère d'évaluation peut être la somme des étiquettes des contraintes satisfaites et celles des contraintes consistantes entre elles d'une branche de l'arbre. Un tel algorithme doit explorer l'arbre de recherche tout en espérant que la solution soit du côté gauche de l'arbre (c.à.d espérer obtenir la borne optimale après un petit nombre d'appels à Houria).

Ce qui peut être aussi intéressant à faire dans cette voie, c'est la conception d'une version incrémentale de cet algorithme ainsi que l'intégration et l'utilisation d'autres critères globaux de comparaison utilisant l'erreur métrique. Ceci doit permettre d'une manière incrémentale la satisfaction des buts intermédiaires comme ceux créés par des applications graphiques interactives.

8 Latif : résolveur d'une hiérarchie de contraintes à sorties multiples utilisant plusieurs critères.

Ce chapitre décrit une autre partie de notre travail. On s'intéresse à la coopération de différents résolveurs afin de résoudre une hiérarchie ayant les contraintes de son dernier niveau associées à un comparateur global et celles des niveaux supérieurs associées à un critère local. Une des approches prometteuses est celle qui utilise les techniques de la propagation locale pour partitionner un système de contraintes en des sous-ensembles de contraintes qui sont résolus par ces différents résolveurs.

Dans ce chapitre, on s'intéresse au cas où il est nécessaire de résoudre les contraintes simultanément (c.à.d. contraintes formant un cycle et/ou un conflit). On s'intéresse également au cas où les modes de combinaison peuvent varier selon les niveaux. Par exemple, dans une hiérarchie contenant trois niveaux d'importance, on peut utiliser pour les deux premiers niveaux un critère local pour lequel une solution sera meilleure qu'une autre si elle satisfait un sur-ensemble des contraintes satisfaites par la seconde. Pour le troisième niveau, un comparateur global plus fin, comme la somme des carrés des erreurs, qui impose un ordre total sur les configurations, peut être utilisé pour départager les solutions. Ce troisième niveau, utilisant des erreurs numériques, ne peut plus être résolu par propagation locale.

Des travaux précédents sur ce problème avaient abouti à la réalisation d'un algorithme (nommé LSCS [HKS+94]) pour la résolution d'une hiérarchie de contraintes dont les méthodes ne possèdent qu'une seule variable de sortie. Les hiérarchies manipulées par l'algorithme initial utilisé consistent en des hiérarchies ayant au dernier niveau des contraintes associées au comparateur global *Moindres-Carrés-Métrique*, et aux niveaux précédents, des contraintes associées au comparateur *Localement-Prédicat-Meilleur*.

Notre recherche s'est orientée vers la généralisation de cet algorithme. Cette généralisation présente des nouvelles définitions et une extension du mécanisme initial utilisé dans LSCS, ceci dans le but de pouvoir prendre en compte des contraintes possédant des méthodes recalculant plusieurs variables à la fois, ainsi que de piloter différents résolveurs spécifiques aux critères utilisés.

Ce chapitre est composé de quatre parties dont chacune décrit en parallèle l'existant et notre apport pour la généralisation[1]. Dans la première partie, on montre *via* des exemples le besoin de résoudre les contraintes simultanément ainsi que celui de résoudre les contraintes ayant des méthodes possédant plusieurs variables de sortie. On finira cette première partie par une vue générale sur le résolveur que nous proposons (nommé Latif).

La deuxième partie décrit des outils de construction d'un graphe admissible. La troisième partie décrit le processus à utiliser pour obtenir un graphe solution à partir d'un graphe admissible. Cette partie décrit également un exemple pour bien illustrer ce mécanisme. La quatrième partie décrit l'algorithme généralisé pour la résolution d'une hiérarchie lors de l'ajout ou du retrait d'une contrainte.

8.1 Motivations et vue générale sur Latif

8.1.1 Cycles et conflit

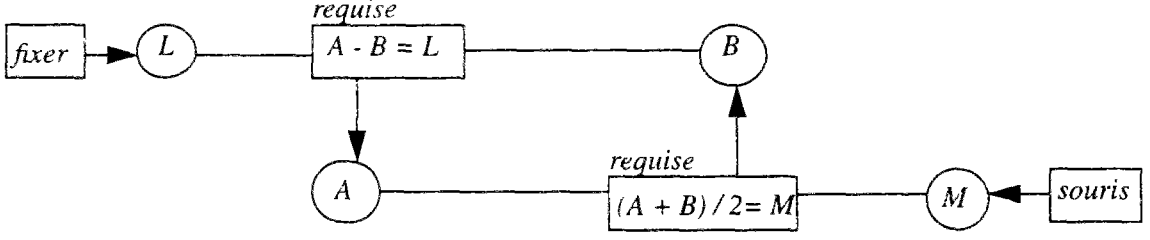
L'algorithme de la propagation locale est très efficace pour des hiérarchies utilisant un critère de comparaison basé sur l'erreur prédicat et ne formant pas de circuits de méthodes des contraintes qui la compose. Malheureusement, beaucoup de hiérarchies du monde réel contiennent des contraintes où leurs méthodes forment des circuits. Ces hiérarchies utilisent aussi des critères basés sur l'erreur métrique. Ces deux cas ne peuvent pas être résolus par la propagation locale. Par exemple, considérons le système de contraintes de la figure 46. Ce système représente une situation typique où le point-milieu *M* d'un segment ayant pour extrémités les points *A* et *B* est déplacé avec une souris. Le point *M* est marqué par la notation *écriture-seulement* et *L* (qui est la longueur du segment *[A,B]*) est marqué par la notation *lecture-seulement* (car on souhaite que cette longueur reste fixe). Pour avoir une solution à ce problème, on est ramené à résoudre un circuit de méthodes. La figure 47 illustre ce fait.

FIGURE 46 : Modélisation par contraintes du déplacement d'un segment par la souris

requise A - B = L?

requise (A + B) / 2 =M!

FIGURE 47 : Graphe de méthodes des contraintes du déplacement d'un segment



Un autre exemple de hiérarchie ne pouvant pas être résolue par la propagation locale est celle de la figure 48 où le comparateur *Moindres-Carrés* est utilisé. Le graphe solution de cette hiérarchie contient un conflit comme le montre la figure 49. Puisque le comparateur *Moindres-Carrés* utilise l'erreur métrique, la solution à cette hiérarchie est $x = 2$.

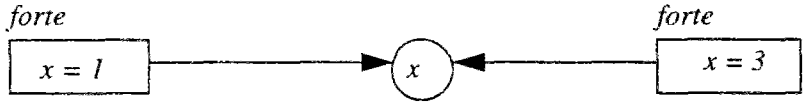
1. Pour des raisons de compréhension, nous avons préféré faire une présentation en parallèle de ce qui existe et de la généralisation.

FIGURE 48 : *Hiérarchie de contraintes conflictuelles.*

forte x = 1

forte x = 3

FIGURE 49 : *Graphe de méthodes d'une hiérarchie de contraintes conflictuelles*



D'une manière générale :

- lors de l'utilisation d'un comparateur basé sur l'erreur métrique, les méthodes des contraintes formant un ou plusieurs conflits doivent être résolues ensemble.
- lors de l'utilisation de n'importe quel critère, si les méthodes des contraintes forment un circuit, alors ces méthodes doivent être résolues ensemble.

8.1.2 Le besoin de considérer des contraintes ayant des méthodes multi-sorties.

Une méthode d'une contrainte est dite multi-sorties si elle détermine plusieurs variables. L'utilisation de ce type de contrainte est considérée pour la résolution des problèmes de cycles. Elle est considérée aussi pour une meilleure modélisation de certaines applications comme par exemples TRIp et TRIpII [KK91,SSA91] qui sont deux applications d'interfaces graphiques. Plusieurs articles de la littérature présentent les avantages et la facilité d'utilisation des contraintes ayant plusieurs variables de sortie [ROS94, San94, Hil93].

Généralement, la modélisation par des contraintes ayant des méthodes multi-sorties a pour but de donner un aspect élégant pour l'expressivité des contraintes du problème traité. Par exemple, considérons les variables x et y qui représentent les coordonnées cartésiennes d'un point, et les variables ρ et θ qui représentent les coordonnées polaires de ce même point. Pour garder ces deux représentations consistantes, on peut définir une contrainte possédant deux méthodes à deux-sorties qui sont m_{1c1} dans un sens et m_{2c1} dans l'autre sens définies dans la figure 50.

FIGURE 50 : *Relation entre coordonnées polaires et cartésiennes d'un point.*

$m_{1c1} = (x,y) \leftarrow (\rho \cos\theta, \rho \sin\theta)$.

$m_{2c1} = (\rho,\theta) \leftarrow \left(\sqrt{x^2 + y^2}, \arctan(y,x)\right)$.

Les méthodes multi-sorties sont aussi utilisées pour accéder aux éléments des structures de données composées. Par exemple, pour décomposer l'objet composé pointCartesien en deux variables, on utilise la contrainte contenant les deux méthodes m_{1c1} et m_{2c1} définis dans la figure 51.

FIGURE 51 : *Décomposition d'un objet composé.*

(x, y)← (point . x, point . y).

point ← Créerpoint(x, y).

8.1.3 Vue générale sur Latif

L'algorithme que nous allons présenter dans ce chapitre est une extension de LSCS présenté dans [HKS+94]. Notre algorithme résout une hiérarchie de contraintes multi-directionnelles et multi-sorties, et de plus, manipule différents types de contraintes.

On s'intéresse au cas où les modes de combinaison peuvent varier selon les niveaux. Par exemple, dans une hiérarchie contenant *n* niveaux d'importance, on peut utiliser pour les *n-1* premiers niveaux un critère local, et pour le $n^{iéme}$ niveau un critère global plus fin qui impose un ordre total sur les configurations pour départager les solutions. Partant d'ici, les hiérarchies pouvant être traitées par Latif sont celles ayant à leur dernier niveau des contraintes associées au comparateur global *Moindre-Carrés-Métrique* ou au comparateur global *Pire-Cas-Métrique* et celles des niveaux supérieurs sont associées à un comparateur local.

Latif peut être vu comme un système de pilotage de résolveurs spécifiques pour le traitement d'une hiérarchie contenant plusieurs types de contraintes. Chacun de ces résolveurs est basé sur un comparateur bien déterminé.

Pour résoudre une hiérarchie de contraintes, cet algorithme prend en compte le fait que certaines des contraintes de cette hiérarchie doivent être résolues ensemble pour former des cellules de contraintes. Dans ce cas, on parle aussi de subdivision de l'ensemble des contraintes de la hiérarchie en des sous ensembles de contraintes qui doivent être résolus séparément. On pourrait tout résoudre par une optimisation globale, mais ici, on espère qu'en traitant séparément chaque cellule, on arrivera à réduire la complexité de la résolution.

L'ensemble des cellules formé par cet algorithme ne doit pas contenir de dépendance cyclique entre les cellules. Donc, on doit définir un ordre partiel entre les cellules de cet ensemble qui permette de propager les valeurs des variables d'une cellule à une autre.

Chacune de ces cellules formées est résolue par un ou plusieurs résolveurs selon les types de contraintes qui la composent. Les valeurs des variables ainsi obtenues à la suite de cette résolution sont propagées à d'autres cellules de contraintes non encore résolues. Cette propagation se fait par l'algorithme de propagation locale.

Il s'agit d'un résolveur incrémental : les contraintes peuvent être ajoutées ou retranchées du système. A chaque ajout ou retrait de contrainte, un nouveau graphe solution de cellules de contraintes est formé et la résolution locale de ces cellules est exécutée. La formation d'une cellule est soumise à des conditions qui favorisent la planification d'un graphe admissible. Des conditions supplémentaires sont ajoutées pour générer un graphe solution à partir du graphe admissible. Ce graphe solution obtenu permet de trouver les solutions *S* après résolution des cellules qui le composent.

Les graphes admissibles et les graphes solutions créés par cet algorithme ont une sémantique légèrement différente de celle des graphes admissibles et des graphes solutions obtenus par les autres résolveurs présentés dans les autres chapitres (il est à rappeler que ces résolveurs sont des algorithmes de propagation locale et que chacun d'eux implémente un critère local ou global utilisant l'erreur prédicat). Cette différence se concrétise par les faits suivants :

un graphe admissible créé par les algorithmes de propagation locale est un graphe qui contient toutes les contraintes requises actives (satisfaites), tandis qu'un graphe admissible créé par Latif est un graphe de cellules contenant toutes les contraintes de la hiérarchie.

un graphe solution créé par les autres résolveurs est un graphe, qui respecte le critère local ou le critère global utilisé, tandis qu'un graphe solution créé par Latif est un graphe qui respecte, le critère global utilisé dans le dernier niveau de la hiérarchie, et le critère local utilisé pour les niveaux supérieurs de la hiérarchie.

Il faut signaler que cet algorithme suppose que chaque contrainte portant sur un ensemble de variables de taille n et ayant m variables en sortie ($n>m$) possède un ensemble de méthodes de taille C_n^m (c.à.d. n'importe quelles m variables de cet ensemble peuvent être déterminées par les n-m variables qui restent[1]).

La résolution d'une cellule de contraintes peut nécessiter un appel à un ou plusieurs résolveurs si cette dernière contient plusieurs types de contraintes. Par exemple, si l'on considère une hiérarchie à trois niveaux *(fort, moyenne, faible)* de contraintes, que les deux premiers niveaux sont associés au comparateur *Localement-Prédicat-Meilleur* (τ_{LPM}), et que le troisième niveau est associé au comparateur *Pire-Cas-Métrique* (τ_{PCM}), alors une cellule du graphe solution peut contenir ces deux types de contraintes. La résolution de cette cellule consiste à appeler un résolveur spécifique sachant résoudre les deux types de contraintes.

8.2 Cellules de contraintes et graphe admissible

Un algorithme de propagation locale échoue à résoudre un graphe de méthodes de contraintes contenant des circuits (ou des conflits si l'erreur métrique est utilisée), puisque tout simplement la technique de propagation locale n'est pas conçue pour cet effet. Pour considérer ce problème, une nouvelle définition de graphe admissible de contraintes uni-sortie a été proposée dans [HKS+94]. Avant de présenter cette définition, on présente d'abord celles qui modélisent un graphe de contraintes et une cellule de ce graphe de contraintes.

8.2.1 Graphe

Définition 8.1

Soit une hiérarchie de contraintes $H=(V,C)$. Le graphe biparti $G=(V, C, E)$ (V et C étant des ensembles de sommet et E est un ensemble d'arêtes) est un graphe de contraintes de H si et seulement si : $E=\{(v, c) \in V \times C / v$ est contrainte par $c\}$.

8.2.2 Cellules de contraintes

Dans [HKS+94] la notion de cellule de contraintes à été introduite. Cette notion est modélisée par la définition suivante :

1. Dans le cas particulier où $m=1$, cette hypothèse est équivalente à celle de l'algorithme écrit dans [18].

Définition 8.2[1]

Soit une hiérarchie de contraintes $H=(V,C)$ et soit $G=(V, C, E)$ un graphe de contraintes de H. pour $X \subseteq V$, Γ est défini par :

$$\Gamma(X) = \{ c \in C / \exists v \in X \wedge (v,c) \in E \}$$

La paire $p=(V_p, C_p)$ est une cellule de contraintes dans G si et seulement si l'une des deux conditions suivantes est vraie :

1. $V_p \subseteq V$, $C_p = \varnothing$ et $Card(V_p) = 1$.
2. $V_p \subseteq V$, $C_p \subseteq C$, *le sous graphe induit par* V_p *et* C_p *est connexe et*

$\forall X \subseteq V_p \; Card(X) \leq Card(\Gamma_p(X))$ *avec* $\Gamma_p(X) = \Gamma(X) \cap C_p$.

La paire $p=(V_p, C_p)$ est une cellule sur-contrainte dans G si et seulement si : $\text{Card}(V_p) < \text{Card}(C_p)$.

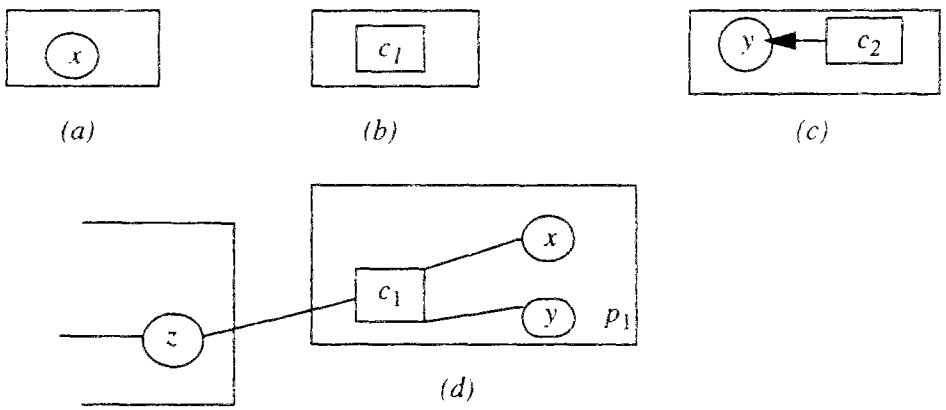
Explicitement, cette définition veut dire qu'on possède une cellule de contraintes dans l'un des deux cas suivants :

1. Si on a une seule variable.
2. Le cardinal de chaque sous-ensemble de variables dans la cellule est inférieur ou égal au nombre de contraintes attachées à ce sous-ensemble de variables dans cette cellule[2].

Dans le cas où le nombre de contraintes dans la cellule p est supérieur au nombre de variables dans p alors, cette cellule est dite sur-contrainte (c.à.d. qu'il existe une variable qui est déterminée par plus d'une contrainte).

La figure 52 montre quelques exemples de cellules de contraintes. Le rectangle contenant le cercle dans la partie (a) de la figure montre une cellule contenant une seule variable. La partie (b) montre une cellule contenant une seule contrainte.

FIGURE 52 : Cellules de contraintes



Les valeurs des variables d'une cellule de contrainte sont obtenues par la résolution des contraintes dans la cellule. Par exemple, dans la cellule de la partie (c) de la figure 41, la valeur de la variable y est déterminée par la contrainte c_2 puisque d'après à la définition 8.2, ceci est possible pour les contraintes uni-sortie.

1. La sémantique de cette définition reste identique à celle donnée dans [HKS+94], mais pour des raisons d'homogénéité nous l'avons traduite dans notre formalisme.
2. Intuitivement, ce cas veut dire : la valeur de chaque variable dans X est déterminée par au moins une contrainte dans C_p.

Le deuxième cas de la définition 8.2 ne permet pas de conclure sur l'existence d'une cellule de contraintes lorsque la hiérarchie contient des contraintes multi-sorties. Par exemple, dans la partie *(d)* de la figure 52, si la contrainte c_1 est à 2 sorties alors on n'est ni dans le cas 1 de la définition 8.2 (puisque p_1 n'est pas réduit à une variable) ni dans le cas 2 de la définition 8.2 (puisque le nombre de variables dans p_1 n'est pas inférieur au nombre de contraintes dans p_1) pour pouvoir dire que p_1 est une cellule et qu'elle peut être résolue localement[1]. Partant de là, et dans le but de pouvoir manipuler des contraintes multi-sorties, on présente par la définition 8.3 une nouvelle modélisation de la notion de cellule de contraintes. Cette dernière peut être vue comme une généralisation de la définition précédente. Juste après la présentation de la nouvelle définition d'une cellule, on donnera plusieurs exemples pour bien montrer son intérêt.

Notation et hypothèses :

Pour $c \in C$, on dénote par $m(c)$ le nombre de variables de sortie de la contrainte c et par $n(c)$ le nombre de variables contraintes par c. $(\forall\, c \in C,\ n(c) - m(c) >= 1)$.On suppose que chaque méthode d'une contrainte c du système possède $n(c)$ variables dont $m(c)$ sont des variables de sortie.

Définition 8.3

Soit une hiérarchie de contraintes $H=(V,C)$ et soit $G=(V, C, E)$ un graphe de contraintes de H. Soit $p=(V_p, C_p)$ avec $V_p \subset V$ et $C_p \subset C$.

pour $c \in \Gamma_p(X)$, on dénote par $n_X(c) = Card\ \{v \in X, (v, c) \in G\}$ *(il est à noter que :* $n_X(c) \leq n(c)$*)*.

La paire p est une cellule de contraintes de G si et seulement si l'une des deux conditions suivantes est vraie :

1. $C_p = \varnothing$ et $Card(V_p) = 1$,
2. *le sous graphe de G induit par* V_p *et* C_p *est connexe* et

$\forall\, X \subseteq V_p\ Card(X) \leq \mu(\Gamma_p(X))$ avec $\mu(\Gamma_p(X)) = \sum_{c \in \Gamma p(X)} Inf(m(c), n_X(c))$.

Explicitement, cette définition généralisée veut dire qu'on possède une cellule de contraintes dans l'une des deux cas suivant :

1. Si on a une seule variable.

2. Le cardinal de chaque sous ensemble de variables dans la cellule est inférieur ou égal à la somme des nombres minimaux de variables liés à ce sous ensemble. Ces nombres sont obtenus à partir des contraintes attachées aux variables de ce sous ensemble, en considérant pour chacune de ces contraintes, ses variables de sortie et ses variables dans ce sous-ensemble.

Exemples de cellules :

La partie (a) de la figure 53 montre le cas où p_1 est une cellule contenant une contrainte uni-sortie selon les deux définitions 8.2 et 8.3, puisque :

Si on considère la définition 8.2, on a la condition du deuxième cas vérifiée car :

$\forall\, X \subseteq V_p\ Card(X) \leq Card(\Gamma(X) \cap C_p) \Leftrightarrow Card\{x\} \leq Card\{c_1\}$

$Card\{x\} \leq Card\{c_1\} \Leftrightarrow 1 \leq 1 \Leftrightarrow$ *vraie.*

Si on considère la définition 8.3, on a la condition du deuxième cas vérifiée car :

$\forall\, X \subseteq V_p\ Card(X) \leq \sum_{c \in \Gamma p(X)} Inf(m(c), n_X(c)) \Leftrightarrow Card\{x\} \leq Inf(m(c_1), 1)$

$Card\{x\} \leq Inf(m(c_1), 1) \Leftrightarrow 1 \leq 1 \Leftrightarrow$ *vraie.*

1 Les variables de sortie d'une contrainte dans la cellule doivent être déterminées au sein de cette cellule.

La partie (b) de la figure 42 montre le cas où p_1 peut être une cellule par la définition 8.3, tandis qu'avec la définition 8.2 elle ne l'est pas.

Si on considère la définition 8.2, p_1 n'est pas considérée comme cellule puisque :

$$\forall X \subseteq V_p \; Card(X) \leq Card(\Gamma(X) \cap C_p) \Leftrightarrow Card\{x,y,z\} \leq Card\{c_1\} \Leftrightarrow 3 \leq 1 \Leftrightarrow faux.$$

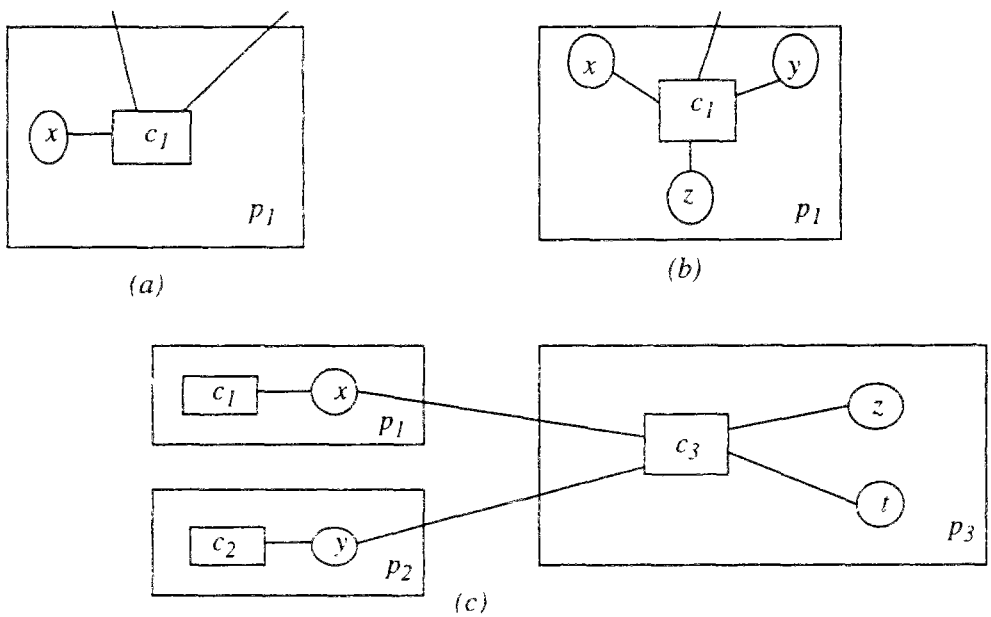
Si on considère la définition 8.3 on a la condition du deuxième cas vérifiée lorsque la contrainte c_1 est à 3 sorties[1]. Si c_1 a deux ou une variable en sortie, alors p_1 ne sera pas considéré comme une cellule.

$$\forall X \subseteq V_p \; Card(X) \leq \sum_{c \in \Gamma p(X)} Inf(m(c), n_X(c)) \Leftrightarrow Card\{x,y,z\} \leq Inf(m(c_1), 3)$$

$$Card\{x,y,z\} \leq Inf(m(c_1), 3) \Leftrightarrow 3 \leq Inf(3,3) \Leftrightarrow vraie.$$

La partie (c) de la figure 42 montre le cas d'un graphe contenant p_1, p_2 et p_3. Les contraintes c_1 et c_2 sont uni-sortie et la contrainte c_3 est à 2 sorties. p_1 et p_2 sont considérées comme des cellules par les deux formalismes des deux définitions 8.2 et 8.3. Seul le formalisme de la définition 8.3 considère que p_3 est une cellule.

FIGURE 53 : Cellules de contraintes selon les formalismes définis.



La figure 54 montre le cas d'une cellule contenant plus d'une contrainte selon le formalisme de la définition 8.3. On montre par cet exemple le rôle de la somme (c.à.d $\sum_{c \in \Gamma p(X)} Inf(m(c), n_X(c))$) dans la détermination des variables de sorties des contraintes dans une cellule[2]. On suppose que les deux contraintes c_1 et c_2 sont à 2 sorties. La contrainte c_1 et les variables sur lesquelles elle porte ne constituent pas une cellule (puisque $Card\{x,y,z\} > Inf(2, 3)$). La contrainte c_2 et le sous ensemble de variables $\{y,t\}$ sur lequel elle porte, constituent une cellule de contraintes (puisque $Card\{y,t\} \leq Inf(2, 2)$). Le fait de considérer ces deux contraintes ensemble avec les variables $\{x,y,z,t\}$ constitue une cellule bien formée. Ceci provient de la variable y qui est une variable de sortie de la contrainte c_2 et aussi utilisée comme variable d'entrée de la contrainte c_1. Par conséquent, l'utilisation de la somme est justifiée. Lorsque les variables e et f seront déterminées par d'autres contraintes dans d'autres cellules, leurs valeurs seront propagées à la cellule p_1.

1. Dans la phase de résolution, les trois variables seront déterminées par la contrainte c_1 dans la cellule.
2. La définition 8.3 telle qu'elle est conçue, permet la détermination des variables de sortie localement dans la cellule.

La contrainte c_2 pourra ainsi déterminer les valeurs de ses deux variables de sortie t et y et ensuite la contrainte c_1 déterminera les valeurs de ses variables de sortie x et z.

La condition $\forall X \subseteq V_p$ $Card(X) \leq \Sigma_{c \in \{c1,c2\}} Inf(m(c), n_X(c))$ est vraie puisque :

Si $X=V_p$ on aboutit à $4 \leq (Inf(2,3) + Inf(2,2))$ ce qui est vrai.

Si $X=\{x,y\}$ on aboutit à $2 \leq (Inf(2,2) + Inf(2,1))$ ce qui est vrai.

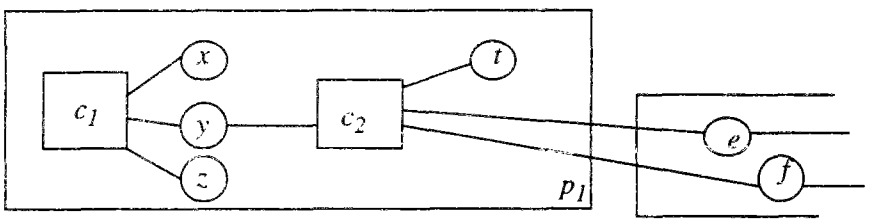
Si $X=\{x\}$ on aboutit à $1 \leq (Inf(2,1) + Inf(2,0))$ ce qui est vrai.

...

pour tout ensemble dans l'ensemble des parties de $\{x,y,z,t\}$ on obtient vrai.

D'une façon générale, lorsqu'un sous graphe induit par n contraintes et leurs variables est connexe et la condition $\forall X \subseteq V_p$ $Card(X) \leq \Sigma_{c \in \Gamma p(X)} Inf(m(c), n_X(c))$ est réalisée alors on a une cellule contenant ces n contraintes.

FIGURE 54 : Cellule contenant plus d'une contrainte



Remarque 8.1:

Si toutes les contraintes dans C sont à une sortie $(c.à.d.\ m(c)=1)$ alors la définition 8.3 est équivalente à la définition 8.2 du fait que $\mu(\Gamma_p(X)) = Card(\Gamma_p(X))$ puisque :

$\mu(\Gamma_p(X)) = \Sigma_{c \in \Gamma p(X)} Inf(1, n_X(c))$

$= \Sigma_{c \in \Gamma p(X)} 1$ puisque $n_X(c)$ est toujours supérieur ou égal à 1

$= Card(\Gamma_p(X))$

Propriété 8.1:

$\Gamma_p(V_p) = C_p$ avec $(p \neq (V_p, \varnothing))$

Preuve 8.1:

- $\Gamma(V_p) = \{ c \in C / \exists v \in V_p$ et $(v,c) \in E \}$

- p est connexe $\Rightarrow (V_p=\varnothing$ et $Card(C_p)=1)$ ou $(C_p \subset \Gamma(V_p))$

par définition : $\Gamma_p(V_p) = \Gamma(V_p) \cap C_p$, et donc, puisque $C_p \subset \Gamma(V_p)$ alors $\Gamma_p(V_p) = C_p$.

Propriété 8.2 :

La condition $Card(V_p) \leq \mu(C_p)$ (c.à.d $Card(X) \leq \mu(\Gamma_p(X))$) est nécessaire pour avoir une cellule mais elle n'est pas suffisante. Il faut considérer aussi tous les éléments de l'ensemble des parties de V_p (comme il est indiqué dans la définition 8.3).

Preuve 8.2:

Considérons la figure 55, on suppose[1] que toutes les contraintes sont uni-sortie ($m(c_i) = 1$ $\forall i \in [1..5]$).. Soit $p = (V_p, C_p)$ avec $V_p = \{x_1, x_2, x_3, x_4\}$, $Card(V_p) = 4$ et $C_p = \{c_1, c_2, c_3, c_4, c_5\}$.

$\mu(C_p) = \Sigma_{c \in Cp} Inf(1, n_{Vp}(c))$

$= \Sigma^5_{i=1} Inf(1, n_{Vp}(c_i))$

$= \Sigma^5_1 1 = 5$ puisque $n_{Vp}(c_1) = 1$, $n_{Vp}(c_2) = 1$, $n_{Vp}(c_3) = 2$, $n_{Vp}(c_4) = 1$, $n_{Vp}(c_5) = 3$.

La condition $Card(V_p) \leq \mu(C_p)$ est vérifiée (puisque $4 \leq 5$) et pourtant p n'est pas une cellule puisque la condition : $\forall X \subseteq V_p$ $Card(X) \leq \mu(\Gamma_p(X))$ de la définition 8.3 n'est pas vérifiée. Il suffit de prendre :

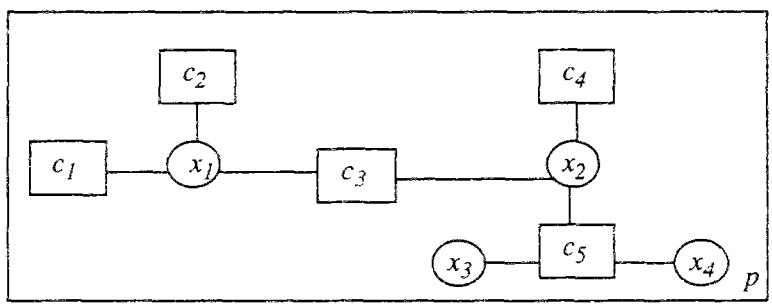
$X = \{x_3, x_4\}$ et $\Gamma_p(X) = \{c_5\}$ et donc on a :

$\mu(\Gamma_p(X)) = Inf(n_X(c_5), m(c_5))$

$= Inf(n_X(c_5), 1)$

$= Inf(2, 1) = 1$. On aboutit donc à : $Card(X) > \mu(\Gamma_p(X))$ (car $2 > 1$).

FIGURE 55 : Graphe connexe ne constituant pas une cellule.



8.2.3 Cellule sur-contrainte

On définit maintenant la notion de cellule sur-contrainte par la définition suivante :

Définition 8.4:

Soit $p = (V_p, C_p)$ une cellule de contraintes telle que $C_p \neq \emptyset$. p est une cellule sur-contrainte si et seulement si : $Card(V_p) < \mu(\Gamma_p(V_p))$.

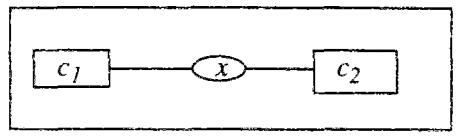
En utilisant la propriété 8.1, la cellule de contrainte p ($\neq (V_p, \emptyset)$) est une cellule sur-contrainte si et seulement si : $Card(V_p) < \mu(C_p)$. Nous verrons par la suite l'utilisation de ce type de cellule.

Exemple :

La cellule de la figure 56 montre une cellule sur-contrainte puisque $Card(Vp) = 1$ et $\mu(C_p) = 1 + 1 = 2$.

1. On peut faire cette supposition puisqu'on a montré que les deux définitions 8.2 et 8 3 sont équivalentes lorsqu'il s'agit de contraintes uni-sortie.

FIGURE 56 : Cellule sur-contrainte



8.2.4 Graphe admissible

Définition 8.5[1]:

Etant donné un graphe de contraintes $G=(V,C,E)$ et un ensemble P de cellules de contraintes dans B. Le quadruplet $G_A=(V,C,E,P)$ est un graphe admissible pour G si et seulement si :

Chaque variable dans V se trouve dans une et une seule des cellules de P.

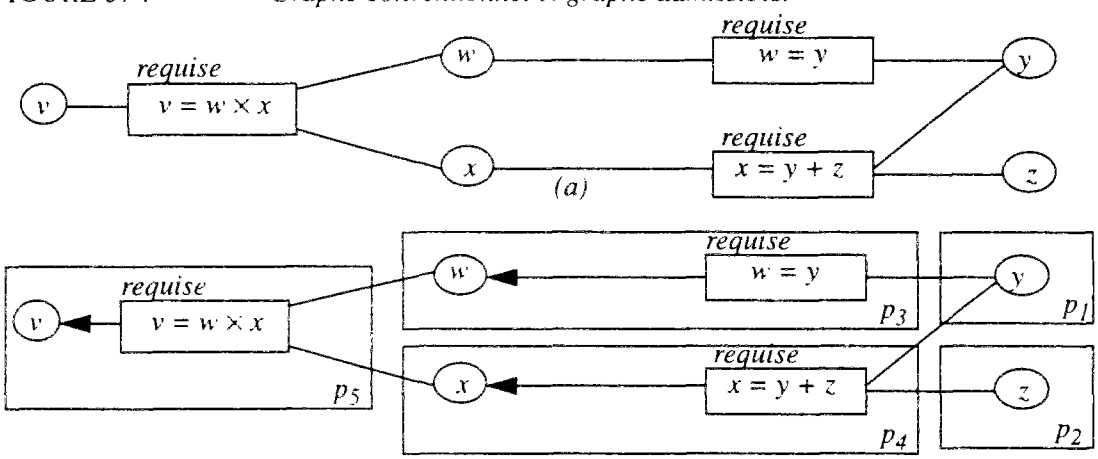
Chaque contrainte dans C se trouve dans une et une seule des cellules de P.

Il n' y a pas de dépendance cyclique entre les cellules de P.

Exemple :

La partie (b) de la figure 57 montre un graphe admissible équivalent à celui de la partie (a) de cette figure. Une valuation qui satisfait les contraintes de cet exemple peut être obtenue en résolvant les cellules: p_1, p_2, p_3, p_4 et p_5 dans cet ordre et en utilisant l'algorithme de la propagation locale pour propager les valeurs des variables entres les cellules.

FIGURE 57 : Graphe conventionnel et graphe admissible.



L'intérêt principal d'un graphe composé de cellules de contraintes est de pouvoir modéliser la prise en compte des circuits entre les méthodes des contraintes du système, et aussi, les conflits des méthodes des contraintes associées à un comparateur utilisant l'erreur métrique. Dans la section suivante, on présentera le mécanisme à utiliser pour obtenir un graphe solution à partir d'un graphe admissible. Les graphes solutions que l'on vise à obtenir ici sont des graphes solutions à des hiérarchies possédant à leurs derniers niveaux des contraintes associées au comparateur global *Moindre-Carrés-Métrique* ou au comparateur global *Pire-Cas-Métrique* (ou encore un autre comparateur qui possède les mêmes propriétés que ces deux derniers).

1. Cette définition est équivalente à celle donnée dans [HKS94+], mais pour ce qui nous concerne ces fonctionnalités sont différentes puisqu'elle repose sur la notion de cellule.

8.3 Graphe solution

Dans la première partie de cette section, on présente un exemple qui montre le déroulement de la procédure visant à l'obtention d'un graphe solution pour un des types des hiérarchies cité auparavant. En deuxième partie de cette section, nous présentons quelques définitions qui permettent d'avoir des outils. Ces derniers serviront à définir d'une manière générale un graphe solution afin de produire une solution à la hiérarchie. Nous rappelons ici qu'une hiérarchie peut contenir plusieurs types de contraintes (c.à.d les niveaux de la hiérarchie excepté le dernier, ne sont pas forcement associés au même comparateur local) avec, en dernier niveau, les contraintes sont associées à l'un des comparateurs *Pire-Cas-Métrique* (type de solution τ_{PCM}), *Moindre-Carrés-Métrique* (type de solution τ_{MCM}).

8.3.1 Exemple d'un graphe solution

Les graphes conventionnels ne peuvent pas produire de solution à une hiérarchie du type décrit auparavant. Par contre les graphes composés de cellules de contraintes sont un moyen pour la résolution de ce type de hiérarchies. Par exemple, considérons la hiérarchie à trois niveaux de préférence décrite dans la figure 58. Les deux niveaux *fort* et *moyen* sont associés au comparateur *Localement-Prédicat-Meilleur* (τ_{LPM}) et le niveau *faible* est associé au comparateur *Moindres-Carrés-Métrique*. Alternativement, on supposera aussi que ce niveau peut être associé au comparateur *Pire-Cas-Métrique*. Les contraintes de cette hiérarchie sont toutes supposées pondérées à un poids de 1. Toutes les contraintes ici sont uni-sortie, exceptée la contrainte c_4 qui est une contrainte à 2 sorties.

FIGURE 58 : Hiérarchie à trois niveaux intégrant deux types de contraintes.

requise

$c_1 : v=0$

$c_7 : t-s=1$

forte

$c_4 : (z, y) = (v-y, x+v)$

$c_8 : t+s=r$

moyenne

$c_6 : r=2$

faible

$c_2 : w=v$

$c_3 : x=1$

$c_5 : y=t$

La figure 59 montre le graphe de contraintes de cette hiérarchie et la figure 60 montre un graphe de méthodes pour cette hiérarchie. Ce dernier ne peut pas produire par propagation locale une solution à cette hiérarchie puisqu'il contient un conflit entre la contrainte c_1 et la contrainte c_2 sur la variable v et un autre conflit entre c_4 et c_5 en y ainsi qu'un circuit entre c_7 et c_8.

FIGURE 59 : Graphe de contraintes

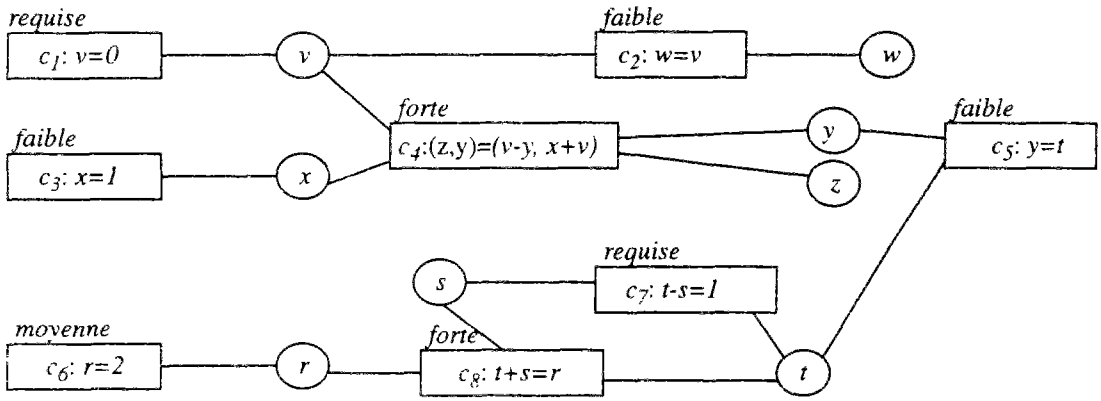
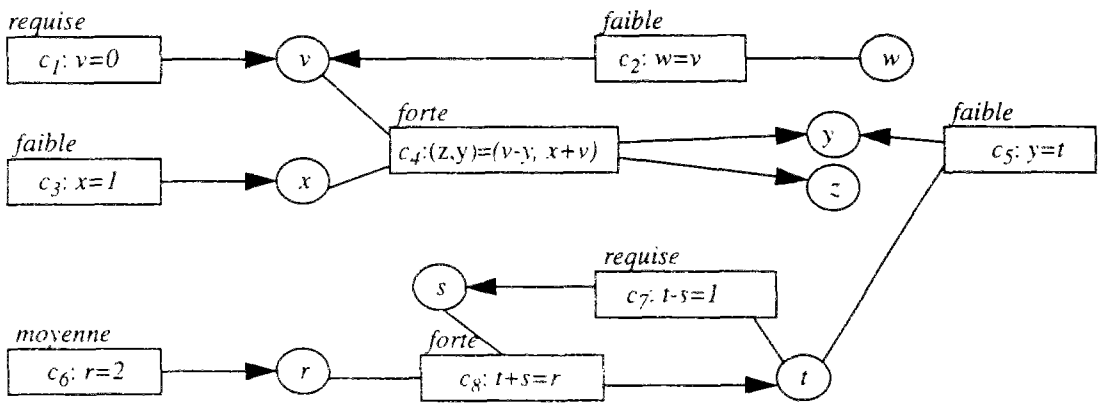


FIGURE 60 : Graphe conventionnel contenant cycle et conflits



Pour débuter, on crée un graphe admissible composé de cellules qui peuvent contenir des cycles et des conflits. Ceci est illustré par la figure 61. En satisfaisant les contraintes localement dans les cellules, on obtient la valuation θ: *{x=1, v=0, w=0, y=1, z=-1, r=2, t=3/2, s=1/2}*. Les séquences de combinaisons d'erreurs par niveau et selon le critère associé sont:

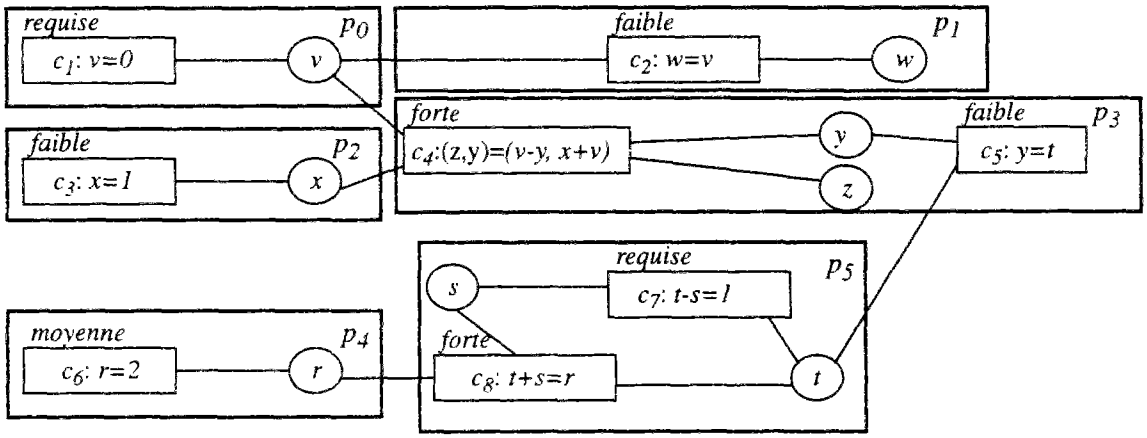
forte : $g_{\tau LPM}(E_{\tau LPM}([c_4,c_8]\ \theta)) = g_{\tau LPM}([0,0])=[0,0]$.

moyen : $g_{\tau LPB}(E_{\tau LPM}([\ c_6]\ \theta)) = g_{\tau LPM}([0])=[0]$.

faible : $g_{\tau MCM}(E_{\tau MCM}([c_2,\ c_3,\ c_5]\ \theta)) = g_{\tau MCM}([0,0,1/2]= 0^2+0^2+1/2^2=1/4$.

$g_{\tau PCM}(E_{\tau PCM}([c_2,\ c_3,\ c_5]\ \theta)) = g_{\tau PCM}([0,0,1/2]= 1/2$.

FIGURE 61 : Graphe admissible composé de cellules de contraintes



Le fusion d'une cellule sur-contrainte avec d'autres cellules non sur-contraintes crée en général[1] un meilleur graphe (c.à.d une meilleure valuation). Ceci vient du fait que la nouvelle cellule obtenue après fusion acquiert plus de liberté pour déterminer les valeurs de ces variable. La fusion des cellules repose sur les critères "cellules contenant des contraintes du dernier niveau de la hiérarchie", "cellules adjacentes" et "cellules formant une dépendance cyclique". On définira par la suite la notion de cellules adjacentes.

Par exemple, la cellule p_3 de la décomposition dans la figure 61 est une cellule sur-contrainte (d'après la définition 8.4). En fusionnant cette cellule avec la cellule p_2, on aboutit à la création de la nouvelle cellule p_{23}. Ceci donne le graphe solution de cette hiérarchie montré par la figure 62. Dans ce cas, les valeurs des variables *v, w, r, t* et *s* ne sont pas changées puisqu'elles sont déterminées par les contraintes des autres cellules ($\neq p_2$ *et* $\neq p_3$). Les valeurs des variables *x, y* et *z* sont redéterminées par les contraintes de la nouvelle cellule p_{23} telles que la contrainte c_4 soit satisfaite[2] *(i)* et que l'erreur métrique produite par les contraintes c_3 et c_5 soit minimale *(ii)*[3] *ou (iii)*[4] avec :

$(i) \Leftrightarrow z = 0 - y \wedge y = x+0$

$(ii) \Leftrightarrow (y-3/2)^2+(x-1)^2$ *est minimale.*

$(iii) \Leftrightarrow Max\ (|y-3/2|, |x-1|)$ *est minimale.*

Pour le cas où τ_{MCM} est utilisé alors :

$(i) \wedge (ii) \Leftrightarrow (x=y=-z) \wedge ((x-3/2)^2+(x-1)^2)$ *est minimale*[5].

$(i) \wedge (ii) \Leftrightarrow x=y=-z=5/4.$

Pour le cas où τ_{PCM} est utilisé alors :

$(i) \wedge (iii) \Leftrightarrow (x=y=-z) \wedge Min[Max(|x-3/2|, |x-1|)].$

$(i) \wedge (iii) \Leftrightarrow (x=y=-z) \wedge Min(|x-3/2|$ *avec* $x \in]-\infty, 5/4])$; $|x-1|$ *avec* $x \in [5/4, +\infty[).$

$(i) \wedge (iii) \Leftrightarrow x=y=-z=5/4$

Normalement, chaque calcul effectué au-dessus doit être fait par un résolveur spécifique selon le critère utilisé par les contraintes et selon la nature des contraintes (c.à.d linéaire, non linéaire ...)

1. On garantit que cette opération dans le pire des cas produit un graphe aussi bon que le graphe initial.
2. Puisque la contrainte c_4 est associée au type de solution τ_{LPM}.
3. Si les contraintes c_3 et c_5 sont associées au type de solution τ_{MCM}
4. Si les contraintes c_3 et c_5 sont associées au type de solution τ_{PCM}
5. Ici la dérivée de cette fonction s'annule lorsque $x = 5/4$.

La nouvelle valuation obtenue en utilisant le type τ_{MCM} ou le type τ_{PCM} pour les contraintes du niveau faible est : $\theta' = \{x=5/4,\ v=0,\ w=0,\ y=5/4,\ z=-5/4,\ r=2,\ t=3/2$ *et* $s=1/2\}$. Les séquences de combinaisons d'erreurs par niveau et selon le critère associé sont :

forte $g_{\tau LPM}(E_{\tau LPM}([c_4,c_8]\ \theta')) = g_{\tau LPM}([0,0])=[0,0]$.

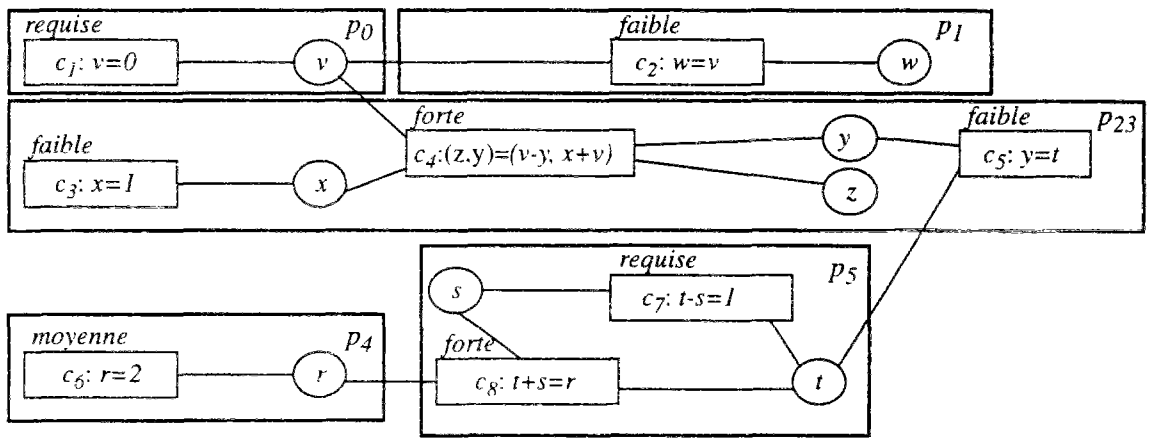
moyen $g_{\tau LPM}(E_{\tau LPM}([\ c_6]\ \theta')) = g_{\tau LPM}([0])=[0]$.

faible $g_{\tau MCM}(E_{\tau MCM}([c_2,\ c_3,\ c_5]\ \theta')) = g_{\tau MCM}([0,1/4,1/4]= 0^2+1/4^2+1/4^2=1/8$.

$g_{\tau PCM}(E_{\tau PCM}([c_2,\ c_3,\ c_5]\ \theta')) = g_{\tau PCM}([0,1/4,1/4]= 1/4$.

Ces séquences montrent bien que θ' est meilleur que θ puisqu'au niveau faible de la hiérarchie, l'erreur après application de θ' est inférieure à celle obtenue après application de θ.

FIGURE 62 : Graphe solution de hiérarchie de contraintes



8.3.2 Comment obtenir un graphe solution

Dans ce paragraphe, on définit comment obtenir un graphe de cellules solution pour une hiérarchie. Ce graphe solution produira une solution à la hiérarchie de contraintes. Cette solution sera obtenue en résolvant localement chaque cellule et en utilisant la propagation locale pour propager les valeurs des variables d'une cellule aux cellules voisines.

Avant de définir un graphe solution, on présente successivement les définitions de cellules adjacentes, de l'*étiquette-interne* d'une cellule et enfin celle de l'*étiquette-voyageuse* d'une cellule. Ces trois définitions sont équivalentes à celles données dans [HKS+94] puisque leurs sémantiques restent valides pour notre problématique, cependant elles sont traduites dans notre formalisme pour une question d'homogénité. Ces trois définitions participent à la définition d'un graphe solution d'une hiérarchie de contraintes que nous allons concevoir.

La définition de l'*étiquette-voyageuse* d'une cellule est inspirée de celle introduite dans les formalismes DeltaBlue et SkyBlue pour les variables (voir chapitre 4).

Cellules adjacentes

Définition 8.6:

Soient p et p' deux cellules de contraintes telles que $p=(V_p, C_p)$ et $p'=(V_{p'}, C_{p'})$. *p' est adjacente à p* si et seulement si $\exists l>=1\ \exists v_1,...,v_l \in V_{p'}\ \exists c \in C_p$ avec : $(v_1, c) \in E,....,(v_l, c) \in E$.

Explicitement, une cellule est adjacente à une autre cellule si elle possède au moins une variable reliée à une contrainte dans cette autre cellule. Intuitivement, le but de cette définition est : lorsqu'on fusionne une cellule sur-contrainte avec certaines cellules voisines adjacentes à cette cellule[1], on est certain que la nouvelle cellule obtenue produira une solution meilleure (ou au moins aussi bonne) que la solution obtenue par résolution de ces cellules sans les fusionner.

Exemple :

Reprenant l'exemple de la figure 61, les cellules p_0 , p_2 et p_5 sont adjacentes à la cellule p_3. Ici, seule la cellule p_2 est fusionnée à la cellule p_3 (puisqu'il est impossible de créer une solution meilleure si l'on fusionne aussi p_0 et/ou p_5).

On verra *via* les définitions de l'*étiquette-interne* et de l'*étiquette-voyageuse* d'une cellule, comment on peut prédire (ou déterminer) l'ensemble de cellules adjacentes à une cellule sur-contrainte dans un graphe admissible qu'il faut fusionner pour obtenir un graphe solution.

Etiquette-interne d'une cellule de contraintes

Définition 8.7:

Soit $p=(V_p, C_p)$ une cellule de contraintes. Si $C_p = \varnothing$ alors l'*étiquette-interne* d'une cellule est égale à *très-faible*[2] sinon elle est égale à la plus faible valeur de l'ensemble des valeurs des étiquettes associées aux contraintes dans C_p.

Exemple :

Reprenons l'exemple de la figure 61. L'*étiquette-interne* de la cellule p_3 est égale à *faible* (puisque *Inf(faible, forte) = faible*).

Etiquette-voyageuse d'une cellule de contraintes

Définition 8.8 :

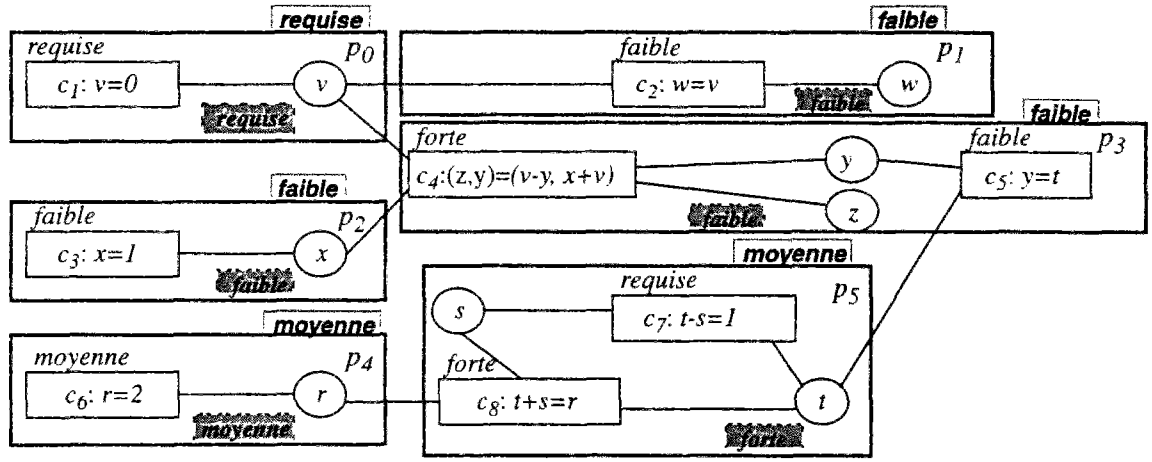
Soit $p=(V_p, C_p)$ une cellule de contrainte. L'*étiquette-voyageuse* de p est égale à l'étiquette minimale dans l'ensemble formé par l'*étiquette-interne* de p et les *étiquettes-voyageuses* des cellules adjacentes à p (cette définition est récursive, mais on garantit toujours l'existence d'un point fixe pour n'importe quel graphe de cellules).

Exemple :

Si l'on considère l'exemple de la figure 61, si on calcule les *étiquettes-voyageuses*[3] des cellules de ce graphe alors on aboutit au graphe de la figure 53.

1. Nous verrons après pourquoi on a dit : "certaines cellules voisines" et pas "toutes les cellules voisines".
2. On suppose que *tres-faible* est la plus faible étiquette de la hiérarchie.
3. On calcule d'abord les *étiquettes-internes* des cellules afin de pouvoir calculer les *étiquettes-voyageuses*.

FIGURE 63 : *Etiquettes-voyageuses et étiquettes-internes* des cellules d'un graphe



Théorème 8.1

Soit p une cellule sur-contrainte telle que :

> tout couple formé par les contraintes ayant des étiquettes différentes de l'étiquette-interne de cette cellule et les variables de ces contraintes dans cette cellule ne constitue pas une cellule.

Soit p' une cellule adjacente à p.

1. Si (l'*étiquette-interne* de p = l'étiquette du dernier niveau de la hiérarchie = l'*étiquette-voyageuse* de p') alors la résolution de la cellule résultante de la fusion de p et p' produira une valuation meilleure (au moins aussi bonne) que celle obtenue par la résolution séparée de p et de p'.

2. Si (l'*étiquette-interne* de p = l'étiquette du dernier niveau de la hiérarchie < l'*étiquette-voyageuse* de p') alors la résolution de la cellule résultante de la fusion de p et p' ne produira pas de valuation meilleure que celle obtenue par la résolution séparée de p et de p'.

Preuve :

On dénote par i_p l'*étiquette-interne* de p (qui est aussi l'étiquette associée au dernier niveau de la hiérarchie) et par $v_{p'}$ l'*étiquette-voyageuse* de p'. D'après la définition de l'*étiquette-voyageuse*, on affirme que les variables de la cellule p' sont déterminées par des contraintes ayant des étiquettes plus fortes ou égales à $v_{p'}$ *(1)*. D'après l'hypothèse de ce théorème et les définitions d'une cellule sur-contrainte et de l'*étiquette-interne*, on affirme que les contraintes étiquetées par i_p contribuent à déterminer les variables de la cellule p *(2)*.

1. Partant des affirmations *(1)* et *(2)* et puisque i_p est égale à $v_{p'}$, le fait de fusionner p et p' ne peut que faire décroître l'erreur globale des contraintes du dernier niveau de la hiérarchie (c.à.d celle associées à l'un des comparateurs *MCM* ou *PCM*) puisque ces contraintes auront plus de liberté pour déterminer les variables, et de plus , cette fusion n'aura pas d'effet sur un niveau plus haut associé à un critère local. Par conséquent, il y aurait alors création d'une valuation meilleure que celle produite par une résolution séparée des deux cellules p et p'.

2. Partant des affirmations *(1)* et *(2)* et puisque i_p est plus faible que $v_{p'}$ alors aucune contrainte du dernier niveau n'est dans p' (ni dans la cellule adjacente à p', ni dans l'adjacente à l'adjacente à p', etc). Par conséquent, il est impossible d'obtenir une valuation meilleure que celle produite par une résolution séparée des deux cellules p et p'.

Dans la figure 63, les cellules p_0, p_2 et p_5 sont toutes adjacentes à la cellule sur-contrainte p_3. Seule p_2 est fusionnée à p_3 puisque son *étiquette-voyageuse (= faible)* est égale à l'*étiquette-interne* de p_3 *(= faible)*. Le résultat issu de cette fusion est dans la figure 62.

Arrivé à ce stade, on dispose maintenant des outils nécessaires pour définir un graphe solution d'une hiérarchie qui, utilise à son niveau le moins important, un des critères globaux cités auparavant. La définition de graphe solution est présentée dans la section suivante.

Graphe solution d'une hiérarchie

Définition 8.9:

Un graphe admissible est un graphe solution s'il satisfait les deux conditions suivantes :

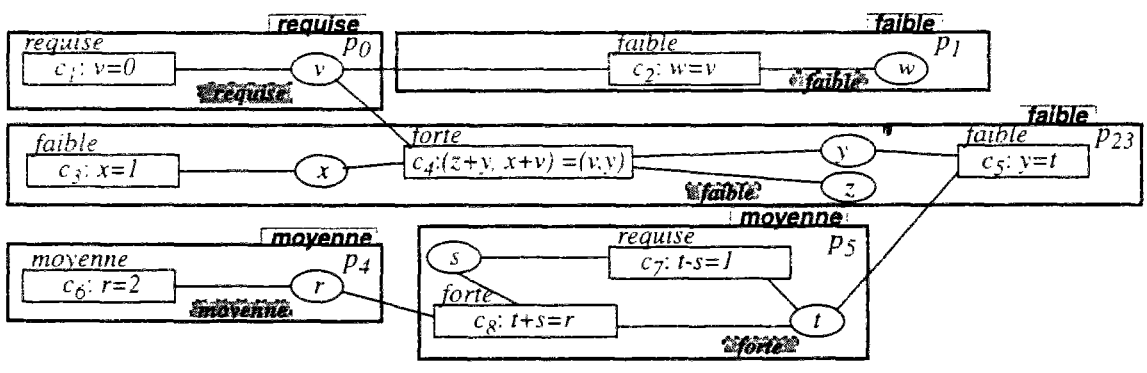
Pour chacune des cellules dans le graphe ayant plusieurs contraintes, le couple formé par les contraintes ayant des étiquettes différentes de l'*étiquette-interne* de cette cellule et les variables de ces contraintes dans cette cellule ne constitue pas une cellule de contraintes.

Pour chacune des cellules *sur-contraintes* contenant des contraintes du dernier niveau de la hiérarchie, son *étiquette-interne* est plus faible que chacune des *étiquettes-voyageuses* des cellules adjacentes à cette cellule sur-contrainte.

La première condition veut dire qu'une cellule de contraintes doit utiliser ses contraintes faiblement étiquetées pour la détermination des valeurs de ses variables. Par conséquent on obtient une valuation produisant une erreur minimale. Par exemple, dans la cellule p_3 de la figure 63, le couple $(c_4, \{y,z\})$ constitue une cellule de contraintes selon la définition 8.3 et par conséquent la première condition de la définition d'un graphe solution n'est pas réalisée. La fusion de p_2 et p_3 en la cellule p_{23} de la figure 64 crée une cellule qui satisfait cette condition (puisque le couple $(c_4, \{x,y,z\})$ ne constitue pas une cellule[1]).

La seconde condition est basée sur le théorème 8.1. Cette condition, si elle est vérifiée, exprimera l'impossibilité de créer un graphe (en fusionnant une cellule sur-contrainte avec d'autres cellules non sur-contraintes) qui produira une valuation meilleure. Les cellules du graphe de la figure 64 obéissent toutes à cette condition puisque la seule cellule sur-contrainte est p_{23} et son étiquette interne *(= faible)* est plus faible que les *étiquettes-voyageuses* des cellules p_0 (= *requise*) et p_5 (= *moyenne*) qui lui sont adjacentes.

FIGURE 64 : Graphe solution de cellules étiquetées



1 Il suffit de prendre $X=\{x,y,z\}$ pour falsifier la condition $\forall X \subseteq \{x,y,z\}\ Card(X) \leq \sum_{c \in \{c4\}} Inf(m(c), n_X(c))$ (car on aboutit à $3 \leq 2$).

8.4 Algorithmes

L'algorithme décrit dans [HKS+94] manipulant uniquement des hiérarchies de contraintes unisortie est conçu en deux couches : un résolveur général et des résolveurs spécifiques. Ici, nous gardons cette structure pour présenter la généralisation que nous avons conçue et qui consiste à considérer les contraintes multi-sorties et de considérer aussi le cas où le comparateur *Pire-Cas-Métrique* est associé au dernier niveau de la hiérarchie.

Le résolveur général planifie un graphe solution et les résolveurs-spécifiques permettent d'obtenir les valeurs des variables en résolvant les contraintes localement dans chaque cellule.

Comme tout algorithme incrémental, Latif est composé de deux procédures *Ajout-contrainte* et *Retrait-contrainte*. La structure utilisée par ces deux procédures est presque la même que celle donnée par l'algorithme initial, cependant les fonctionnalités changent ici puisque les définitions présentées avant opèrent sur un champ un peu plus large du monde des contraintes. La procédure *Ajout-contrainte* invoque deux opérations qui sont : *ajout-variable* et *propagation-locale*. La procédure *Retrait-contrainte* invoque elle aussi deux opérations qui sont : *retrait-variable*, *propagation-locale*. Cette dernière opération utilise l'algorithme de la propagation locale tandis que les autres modifient l'architecture du graphe.

8.4.1 Ajout-contrainte

Initialement, on suppose qu'il existe un graphe solution. Lorsqu'une nouvelle contrainte c est ajoutée à la hiérarchie, la procédure *Ajout-contrainte* met cette nouvelle contrainte dans une nouvelle cellule (ligne 1; figure 65), et cherche la cellule adjacente ayant la plus faible *étiquette-voyageuse* (ligne 2; figure 65).

Si cette *étiquette-voyageuse* est plus faible que l'étiquette de la contrainte c(on est dans le cas où il y a existence d'une ou plusieurs contraintes étiquetées par des étiquettes plus faibles ou égales à l'étiquette de la contrainte ajoutée. Ces contraintes sont appelées contraintes victimes et peuvent être visées afin de garder la sémantique de la hiérarchie. La procédure cherche le chemin dans le graphe des cellules partant de la cellule contenant c et arrivant aux cellules contenant les contraintes victimes en inversant les dépendances entre les cellules dans le chemin[1] (lignes 3,4; figure 65). Après cette opération, la contrainte c devient active.

La procédure élimine les dépendances cycliques entre les cellules générées par la précédente opération en fusionnant les cellules dépendant de la contrainte c (ligne 5; figure 65). Elle met aussi à jour les valeurs des *étiquettes-voyageuses* des cellules du graphe (ligne 6; figure 65).

Si le type de contraintes est l'un des types suivants : τ_{MCM}, τ_{PCM} (ou autre type de comparateur qui se comporte de la même façon que ces deux comparateurs) alors la procédure fusionne la cellule sur-contrainte avec les cellules qui lui sont adjacentes et qui possèdent des *étiquettes-voyageuses* égales à celle de cette cellule sur-contrainte. On rappelle que cela a pour but de minimiser l'erreur des contraintes (cf théorème 8.1) (ligne 7; figure 65).

Finalement, la procédure résout chaque cellule localement en appelant un ou plusieurs résolveurs-types selon les types de contraintes de la cellule et utilise l'algorithme de la propagation locale pour propager les valeurs de cellule en cellule dans le graphe (ligne 8; figure 65).

1. Un pas de cette opération est entièrement décrit dans la figure 66. L'exploration du chemin en entier est réalisée par la boucle tant que.

FIGURE 65 : *Procédure d'ajout d'une contrainte*

Ajout-contrainte *(c , ins-comp-global)*

1 *cl* ← *Cree-cellule(c)*
2 *evm* ← *Min(étiquettes-voyageuses* des cellules adjacentes à *cl)*
3 Si *evm* est plus faible que l'étiquette de *c* alors *eti* ← l'étiquette de *c* Fin-Si
4 Tant que *eti* est plus forte que *evm* faire : ***Chercher-et-décomposer****(cl, eti, evm)* Fin-Tantque
5 Fusionner les contraintes dépendantes de *c* et qui forment un cycle
6 Met à jour les *étiquettes-voyageuses* des cellules dépendantes de *c*
7 Si *((evm* est la plus faible étiquette) ou *(*les contraintes étiquetées par *evm* sont associées à un comparateur dans *ins-comp-global))* alors fusionner les cellules adjacentes à *c* qui possèdent des *étiquettes-voyageuses* égales à celle contenant *c*
8 Résoudre les cellules en utilisant les résolveurs-specifiques et l'opération *propagation-locale*

La figure 66 montre un pas du processus qui inverse la dépendance entre les cellules du chemin partant de la cellule contenant *c* et arrivant à la cellule contenant les contraintes étiquetée par *evm*. La procédure *Chercher-et-décomposer* réalise ce pas. Cette procédure est paramétrée par la cellule courante *cl*, l'*étiquette-interne* de cette cellule courante *eti* et enfin l'*étiquette-voyageuse evm* qui représente l'étiquette des contraintes visées à atteindre.

La procédure repère la cellule adjacente à la cellule *cl* et ayant pour *étiquette-voyageuse evm* (ligne 1; figure 66), et enlève l'ensemble des variables adjacentes à *cl* de la cellule repérée. L'ensemble de ces variables enlevées est ajouté à la cellule *cl* (ligne 3; figure 66)(c.à.d les valeurs de ces variables dépendent maintenant de la contrainte *c*). Après cette opération, la cellule adjacente à *cl* peut devenir sur-contrainte (puisqu'on vient de lui retirer un sous ensemble de l'ensemble de ces variables).

La procédure examine la cellule *adj*, si cette dernière est vide (si elle ne contenait que les variables ôtées) alors l'étiquette *très-faible*[1] est affectée à la variable *eti* afin de pouvoir falsifier la condition de la boucle Tant-que dans la procédure appelante (cf. figure 65).

La procédure compare l'*étiquette-interne* de la cellule *cl* avec l'*étiquette-voyageuse evm* de la cellule adjacente. Si ces deux étiquettes sont égales *(c.à.d* on a atteint l'extrémité du chemin*)* (ligne 5; figure 66) alors la procédure fait appel à une autre procédure pour décomposer[2] cette cellule en un ensemble de cellules (ligne 6; figure 66). La procédure détecte la cellule sur-contrainte dans cet ensemble de cellules obtenu à l'issu de la décomposition (ligne 7; figure 66), et détermine ensuite son *étiquette-interne* (ligne 8; figure 66). Dans le cas où on n'a pas encore atteint l'extrémité du chemin[3] (c.à.d l'*étiquette-interne* de la cellule *cl* n'est pas égale à *evm*) (ligne 9; figure 66) la procédure enlève alors la contrainte de la cellule *adj* (ligne 10,11; figure 66) (Il s'agit ici de la contrainte qui est adjacente à la cellule ayant pour *étiquette-voyageuse evm*).

La procédure décompose la cellule *adj* (ligne 12; figure 66), crée une nouvelle cellule contenant la contrainte enlevée[4] (ligne 13; figure 66) et détermine l'*étiquette-interne*[5] de cette nouvelle cellule crée (ligne 14; figure 66). Enfin, la procédure retourne la position courante du chemin où elle est restée (ligne 15; figure 66).

1. On suppose que *très-faible* est la plus faible étiquette de la hiérarchie.
2. On verra par la suite l'intérêt de cette décomposition.
3. Alors il faut avancer
4. Bien évidemment, cette nouvelle cellule créée a pour cellule adjacente celle qui a l'*étiquette-voyageuse evm*
5. Dans ce cas c'est l'etiquette de la contrainte *cn* de cette cellule

FIGURE 66 : Procédure Chercher-et-décomposer

```
Chercher-et-décomposer(cl, eti, evm)
 1 adj ← la cellule adjacente à cl ayant pour étiquette-voyageuse evm
 2 vars ← l'ensemble des variables dans adj adjacentes à cl
 3 Enlever vars de adj et les remettre dans cl
 4 Si adj = ∅ alors eti ← très-faible
     Sinon
 5     Si l'étiquette-interne de adj est égale à evm alors
 6       cls ← Décomposer (adj, evm)
 7       cl ← une cellule sur-contrainte dans cls
 8       eti ← l'étiquette-interne de cl
       Sinon
10       cn ← la contrainte dans adj telle que cette contrainte est adjacente à une cellule
               ayant pour étiquette-voyageuse evm
11       adj ← adj - cn
12       Décomposer (adj, evm)
13       cl ← une nouvelle cellule contenant la contrainte cn
14       eti ← l'étiquette-interne de cl
       Fin-Si
   Fin-Si
15 Retourner(cl, eti)
```

La procédure *Décomposer* est utilisée pour la véracité de la première condition de la définition d'un graphe solution. Cette procédure décompose une cellule contenant plusieurs contraintes en un ensemble de cellules. Par exemple, considérons le graphe de la figure 67 où les contraintes sont uni-sortie. Ici, il s'agit d'un graphe solution qui est composé d'une seule cellule puisque cette dernière satisfait la condition imposée par la définition d'un graphe solution (la paire $(\{c1,c2,c4\}, \{t,x,y,z\})$ ne forme pas une cellule de contrainte puisque la deuxième condition de la définition 8.3 qui modélise une cellule n'est pas satisfaite du fait que la condition $X=\{t,x,y,z\}$ $Card(X) \leq \sum_{c \in \{c1,c2,c4\}} Inf(m(c), n_X(c))$ est fausse puisqu'on obtient 4≤3).

La figure 68 montre le premier pas réalisé[1] à la suite de l'introduction de la contrainte c_6 (possédant deux variables de sortie t et e). Le fait d'enlever les variables t et e de la cellule p_0 et de les mettre dans la cellule p_1 (figure 69) ne suffit pas pour obtenir un graphe solution, puisque maintenant p_0 ne satisfait plus la première condition imposée par la définition d'un graphe solution (car la paire $(\{c1,c2,c4\}, \{x,y,z\})$ forme une cellule de contrainte selon la définition 8.3, puisque on a la condition $\forall X \subseteq \{x,y,z\}$ $Card(X) \leq \sum_{c \in \Gamma p(X)} Inf(m(c), n_X(c))$ vrai).

Le graphe solution de cette hiérarchie est obtenu en décomposant p_0 en un ensemble de cellules comme cela est montré dans la figure 70. L'idée clé utilisée pour réaliser cette décomposition est de trouver une distribution parfaite des variables sur les contraintes dans la cellule. Cette distribution doit considérer en priorité les contraintes étiquetées par des étiquettes plus fortes que l'*étiquette-voyageuse* de cette cellule[2] (voir exemple plus bas). Par conséquent, on aboutit à la formation de cellules[3] à partir de ces contraintes et ces variables. Comme dans la figure 70, les contraintes étiquetées par *forte* (c.à.d c_1, c_2 et c_4) sont considérées en premier une par une.

1. Création d'une nouvelle cellule contenant cette contrainte et calcul de l'*étiquette-interne* de cette cellule créée
2. Quitte à laisser les contraintes ayant des étiquettes égales à l'*étiquette-voyageuse* non satisfaites.
3. Selon la définition 8.3.

Ensuite, chacune de ces contraintes est associée[1] à une variable[2] liée par cette contrainte et une cellule est formée. On obtient ainsi les trois cellules p_4, p_5 et p_2. Les contraintes restantes c_3 et c_5 forment chacune une cellule qualifiée de sur-contrainte. Ces dernières seront peut-être par la suite fusionnées à d'autres cellules du graphe par la procédure *Ajout-contrainte*.

FIGURE 67 : Graphe solution ayant une seule cellule

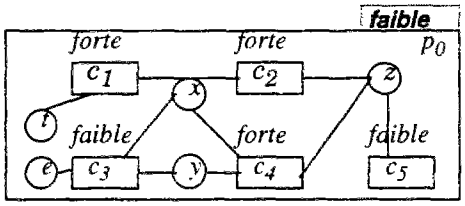


FIGURE 68 : Ajout d'une contrainte à 2-sorties

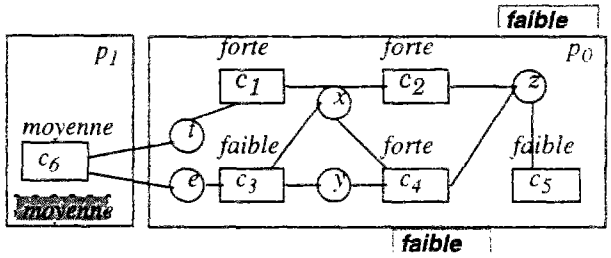


FIGURE 69 : Graphe non solution

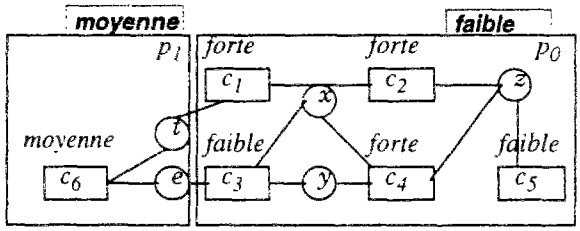
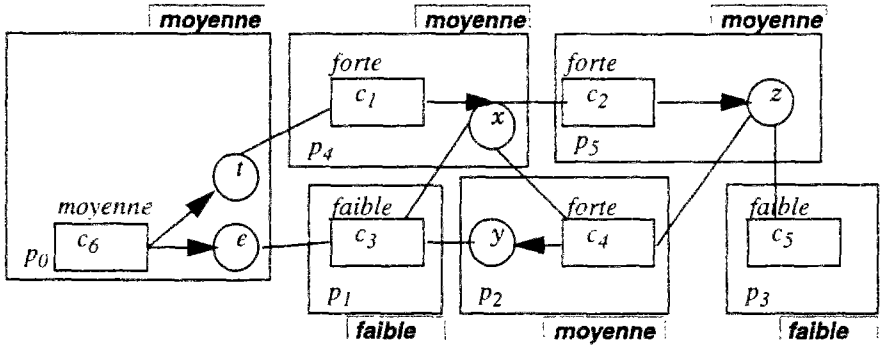


FIGURE 70 : Décomposition d'une cellule et aboutissement au graphe solution



1. Cette association est faite de telle sorte qu'elle soit parfaite. On aurait pu associer la variable x à la contrainte c_3 et la variable z à la contrainte c_5, mais ceci ne constituerait pas une association parfaite puisque la contrainte c_2 n'aurait pas de variable qui lui serait associée. Ou encore, on aurait pu associer les variable x et z à la contrainte c_3 et la variable y à la contrainte c_5, mais ceci ne serait pas bon puisque la paire $(\{x,z\},\{c_3\})$ ne formerait pas une cellule d'après la définition 8.3. etc .
2. Puisqu'on a supposé que ces contraintes sont uni-sortie.

La figure 71 décrit la procédure qui décompose une cellule contenant plusieurs contraintes en un ensemble de petites cellules[1]. Cette procédure est paramétrée par la cellule à décomposer *cl* et par l'*étiquette-voyageuse evm* de cette cellule. Au départ, on isole l'ensemble des contraintes de *cl* telles que chacune de ces contraintes est étiquetée par une étiquette plus forte que *evm* (ligne 1; figure 71). Après cette opération, on construit un ensemble de cellules minimales à partir de l'ensemble des contraintes isolées et l'ensemble des variables dans *cl* (ligne 2; figure 71). Pour chacune des cellules construites, on inverse la dépendance entre contraintes et variables dans cette cellule (ligne 3; figure 71).

La construction de cet ensemble de cellules minimales repose sur une distribution parfaite de l'ensemble des variables[2] dans *cl* à l'ensemble des contraintes isolées et sur la définition 8.3 qui modélise une cellule. La procédure considère ensuite les contraintes restantes dans *cl* (c.à.d ayant des étiquettes équivalentes à *evm*) (ligne 4; figure 71). Pour chacune de ces contraintes, si les variables auxquelles elle est attachée dans *cl* ont été utilisées dans la construction de l'ensemble des cellules minimales, alors une nouvelle cellule serait formée et comporterait cette contrainte (ligne 9; figure 71). Dans le cas contraire, cette contrainte serait considérée avec les variables restantes pour la formation d'une cellule (lignes 7,8; figure 71).

La définition 8.3 qui modélise une cellule de contraintes garantit qu'il ne reste pas de variables indéterminées après la décomposition d'une cellule contenant plusieurs contraintes. Dans le cas où une cellule ne satisfait pas la troisième condition de la définition 8.5 (c.à.d. pas de dépendance cyclique entre les cellules) ou elle ne satisfait pas la deuxième condition de la définition 8.9 (c.à.d. cellule sur-contrainte ayant une cellule adjacente dont l'*étiquette-voyageuse* est plus faible que l'*étiquette-interne* de cette cellule sur-contrainte) cette cellule sera fusionnée avec une cellule du graphe. On prouve que le résultat de cette fusion est une cellule (voir preuve de terminaison ci-dessous).

FIGURE 71 : Décomposition d'une cellule

```
Décomposer(cl, evm)
1 cs ← {c / c ∈ C_cl ∧ étiquette de c est plus forte que evm} .
2 Construire un ensemble de cellules minimales à partir de cs et de V_cl
3 Inverser la dépendance entre contrainte et variable pour chacune des cellules construites
4 cs' ← {c / c ∈ C_cl ∧ étiquette de c est égale à evm}
5 Pour chaque c ∈ cs' faire :
6   S'il reste des variables contraintes par c non utilisées dans le pas 2 alors
7     Crée une cellule à partir de ces variables et de c
8     Inverser la dépendance entre c et les variables dans la cellule crée
    Sinon
9     Crée une cellule contenant c
  Fin-Pour
```

Preuve de terminaison

On peut se poser la question de la terminaison de cet algorithme, c'est-à-dire lors de la fusion de deux cellules formant une dépendance cyclique, est ce que le résultat est une cellule ? La réponse est oui. Dans ce qui suit, on prouvera d'une manière générale que la fusion de deux cellules dont une est adjacente à l'autre résulte en une cellule.

1. Ceci permet aussi une efficacité remarquable lors de la résolution du graphe solution obtenu.
2. Il s'agit de l'ensemble des variables dans *cl* qui sont attachées à ces contraintes.

Soient p et p' deux cellules de contraintes telles que : $p = (V_p, C_p)$ et $p = (V_{p'}, C_{p'})$.

On suppose que $C_p \neq \emptyset$ et $C_{p'} \neq \emptyset$ et que p et p' sont connexes et disjointes entre elles.

Montrons que $p \cup p'$ est une cellule.

Soit $z \subseteq V_p \cup V_{p'}$, on note par $z_p = z \cap V_p$ et $z_{p'} = z \cap V_{p'}$.

On rappelle que $\Gamma(X) = \{c \in C / \exists v \in X \wedge (v,c) \in E\}$ et que $\Gamma_p(X) = \Gamma(X) \cap C_p$.

Il est facile de voir que :

$$\Gamma_{p \cup p'}(z) = \Gamma_p(z_p) \cup \Gamma_{p'}(z_{p'}) \cup ((\Gamma(z_p) \cap C_{p'}) \setminus \Gamma_{p'}(z_{p'})) \cup ((\Gamma(z_{p'}) \cap C_p) \setminus \Gamma_p(z_p)).$$

Et donc :

$$(*)\ \mu(\Gamma_{p \cup p'}(z)) = \sum_{c \in \Gamma p(zp)} Inf(m(c), n_z(c)) + \sum_{c \in \Gamma p'(zp')} Inf(m(c), n_z(c)) + \sum_{c \in ((\Gamma(zp) \cap Cp') \setminus \Gamma p'(zp'))} Inf(m(c), n_z(c)) + \sum_{c \in ((\Gamma(zp') \cap Cp) \setminus \Gamma p(zp))} Inf(m(c), n_z(c)).$$

(dans les formules $Inf(m(c), n_z(c))$ de cette somme on examine $n_z(c)$ car on se place par rapport à $p \cup p'$).

Maintenant il est intéressant de faire les trois remarques suivantes :

(1) $n_{zp}(c) \leq n_z(c)$ puisque $z_p \subset z$ et par conséquent $Inf(m(c), n_{zp}(c)) \leq Inf(m(c), n_z(c))$.

$n_{zp'}(c) \leq n_z(c)$ puisque $z_{p'} \subset z$ et par conséquent $Inf(m(c), n_{zp'}(c)) \leq Inf(m(c), n_z(c))$.

(2) $Card(z_p) \leq \sum_{c \in \Gamma p(zp)} Inf(m(c), n_{zp}(c))$ puisque p est une cellule.

$Card(z_{p'}) \leq \sum_{c \in \Gamma p'(zp')} Inf(m(c), n_{zp'}(c))$ puisque p' est une cellule.

(3) $Card(z) = Card(z_p) + Card(z_{p'})$.

D'après (3) et (2) on aboutit à :

(4) $Card(z) \leq \mu(\Gamma_p(z_p)) + \mu(\Gamma_{p'}(z_{p'}))$

D'après (1) on aboutit à :

(5) $\mu(\Gamma_p(z_p)) + \mu(\Gamma_{p'}(z_{p'})) \leq \sum_{c \in \Gamma p(zp)} Inf(m(c), n_z(c)) + \sum_{c \in \Gamma p'(zp')} Inf(m(c), n_z(c))$.

D'après (*) on a:

(6) $\sum_{c \in \Gamma p(zp)} Inf(m(c), n_z(c)) + \sum_{c \in \Gamma p'(zp')} Inf(m(c), n_z(c)) \leq \mu(\Gamma_{p \cup p'}(z))$.

Par transitivité entre (4) (5) et (6) on aboutit à :

$Card(z) \leq \mu(\Gamma_{p \cup p'}(z))$. *Ceci pour tout* $z \subseteq V_p \cup V_{p'}$. D'où on a bien $p \cup p'$ *une cellule.*

Exemple d'ajout d'une contrainte:

L'exemple de la figure 72 montre le déroulement de la procédure d'ajout d'une contrainte. Initialement, on suppose l'existence d'un graphe solution montré par la figure 72.a. Lorsque la contrainte c_8 est ajoutée à la hiérarchie, la procédure d'ajout décrite avant procède comme suit:

- La cellule p_7 est créée contenant la contrainte c_8 (figure 72.b) et l'étiquette de la contrainte victime est déterminée.

- La variable z est enlevée de la cellule p_6 et ajoutée à la cellule p_7 (figure 72.c).

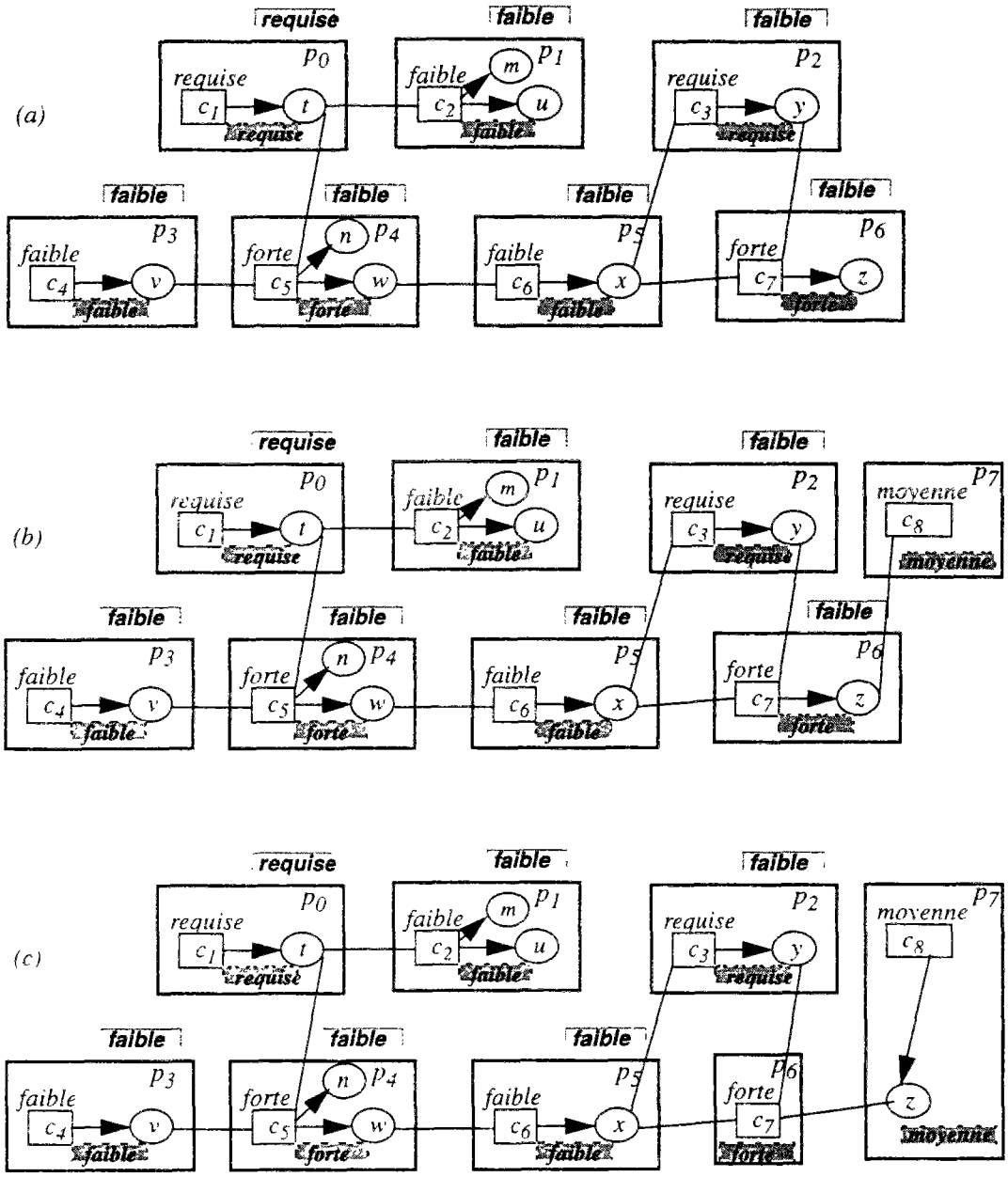
- La variable x est enlevée de la cellule p_5 et ajoutée à la cellule p_6 (figure 72.d). La contrainte c_6 est déterminée à être la contrainte victime.

- Les cellules p_2 et p_6 sont fusionnées puisqu'elles forment une dépendance cyclique (figure 72.e).

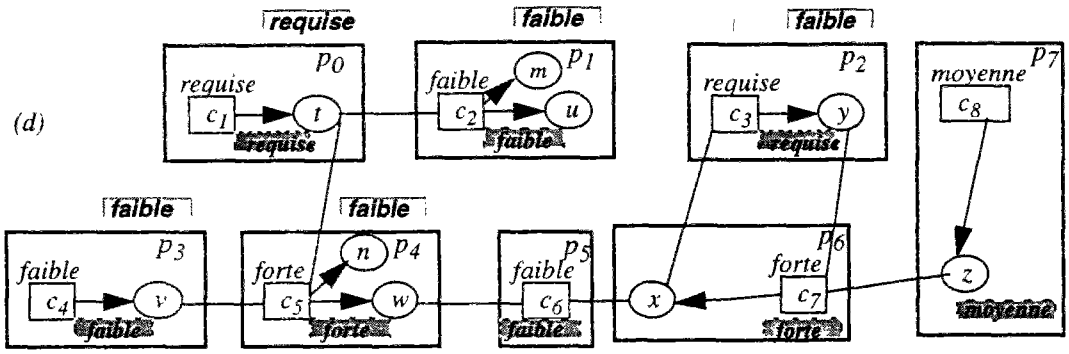
- Les étiquettes-voyageuses des cellules sont mises à jour (figure 72.f).

- Puisque la cellule p_5 est sur-contrainte, elle est fusionnée avec les cellules p_4 et p_3 qui possèdent les mêmes *étiquettes-voyageuses* (figure 72.g).

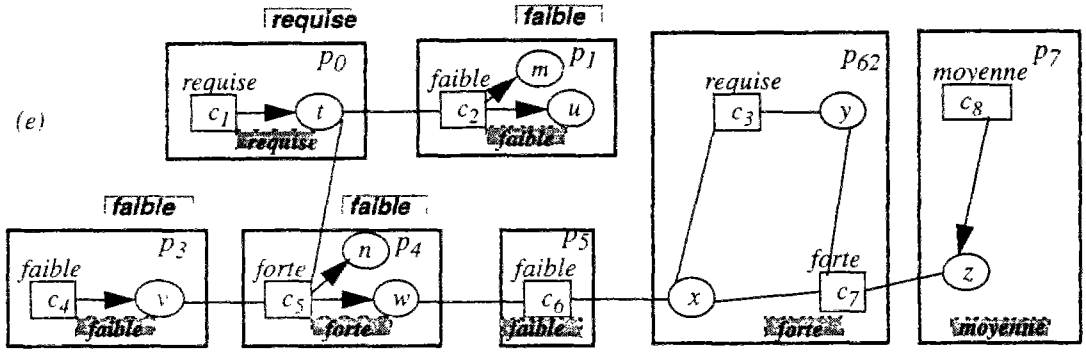
FIGURE 72 : Ajout d'une contrainte à un graphe solution



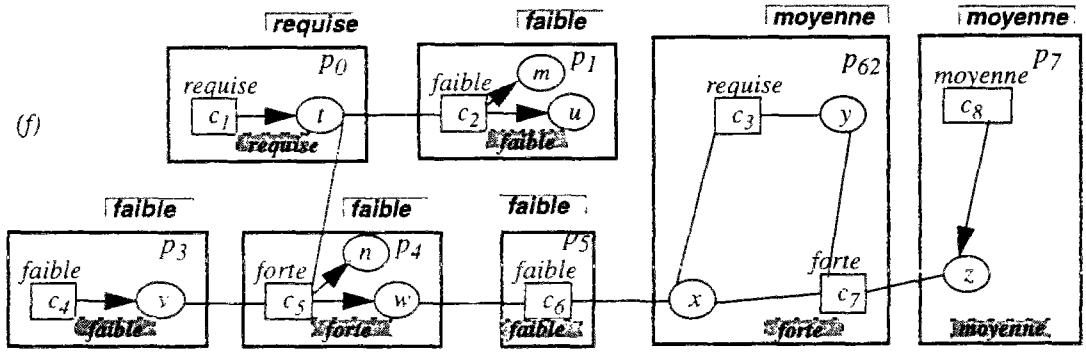
(d)


requise
faible
faible
p0
p1
p2
p7
requise
faible
requise
moyenne
c1
t
c2
m
u
c3
y
c8
faible
faible
p3
p4
p5
p6
faible
forte
faible
forte
c4
v
c5
n
w
c6
x
c7
z
faible
forte
faible
forte
moyenne

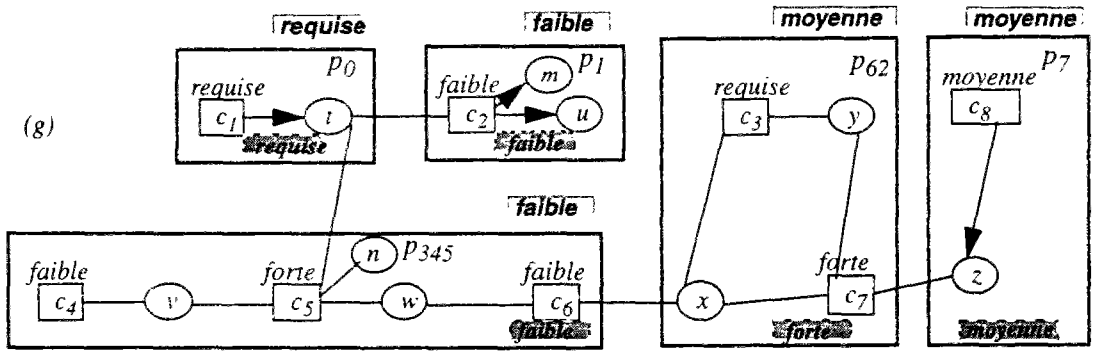
(e)


requise
faible
p0
p1
p62
p7
requise
faible
requise
moyenne
c1
t
c2
m
u
c3
y
c8
faible
faible
p3
p4
p5
faible
forte
faible
forte
c4
v
c5
n
w
c6
x
c7
z
faible
forte
faible
forte
moyenne

(f)


requise
faible
moyenne
moyenne
p0
p1
p62
p7
requise
faible
requise
moyenne
c1
t
c2
m
u
c3
y
c8
faible
faible
faible
p3
p4
p5
faible
forte
faible
forte
c4
v
c5
n
w
c6
x
c7
z
faible
forte
faible
forte
moyenne

(g)


requise
faible
moyenne
moyenne
p0
p1
p62
p7
requise
faible
requise
moyenne
c1
t
c2
m
u
c3
y
c8
faible
p345
faible
forte
faible
c4
v
c5
n
w
c6
x
c7
z
faible
forte
moyenne

8.4.2 Retrait-contrainte

La procédure retrait-contrainte se déroule de la même manière que la procédure ajout-contrainte. Le retrait d'une contrainte de la hiérarchie est réalisé par le retrait de cette contrainte de la cellule où elle se trouve. Ceci peut créer un changement de l'*étiquette-interne* de la cellule et par conséquent un changement de l'*étiquette-voyageuse* de cette cellule (c.à.d que l'on peut être amené à faire décroître l'erreur des contraintes ayant des étiquettes équivalentes, ceci en inversant les dépendances entre la cellule contenant ces contraintes et la cellule qui contenait la contrainte retirée).

L'exemple de la figure 73 montre le déroulement de la procédure du retrait de la contrainte c_8. Lorsque la contrainte c_8 est retirée de la hiérarchie, la procédure de retrait procède comme suit:

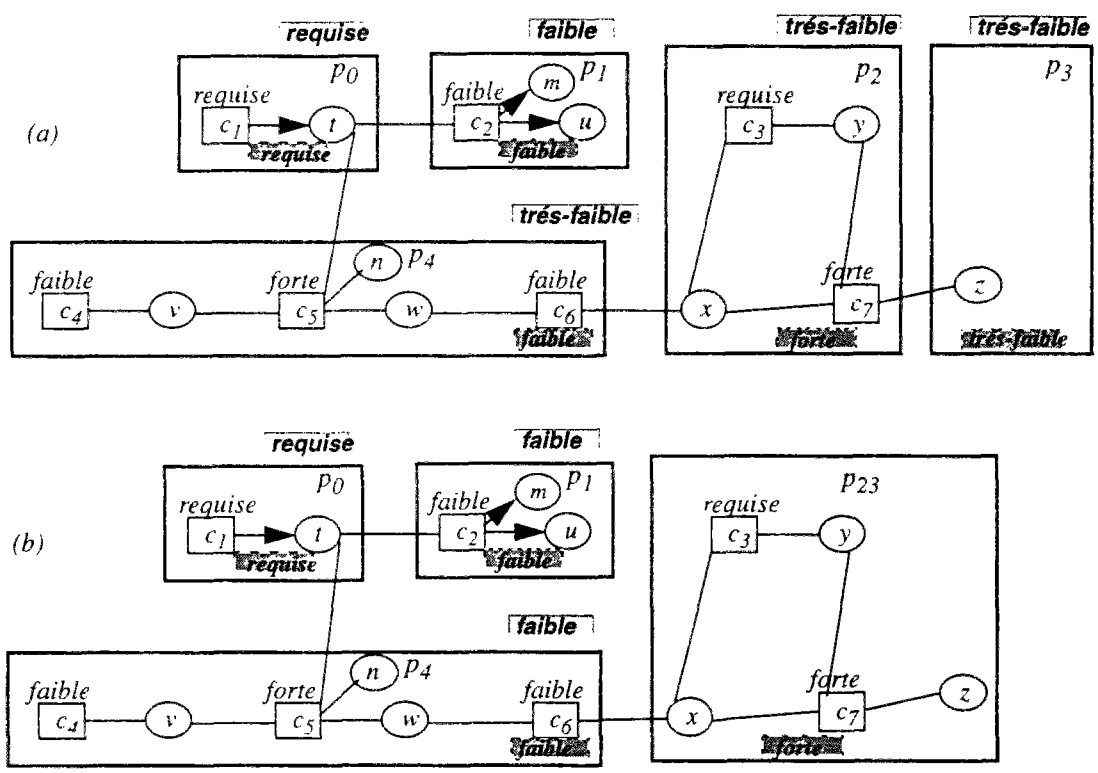
- La contrainte c_8 est retirée de la cellule p_3 et l'*étiquette-interne* de cette cellule est redéterminée (d'après la définition de l'étiquette-interne d'une cellule, cette étiquette a pour valeur *très-faible)*. Les étiquettes-voyageuses des autres cellules sont mises à jour (figure 73.a).

- La cellule p_2 est fusionnée à la cellule p_3 et donc il y a formation de p_{23}. Cette dernière ne constitue pas une cellule d'après la définition 8.3 (figure 73.b).

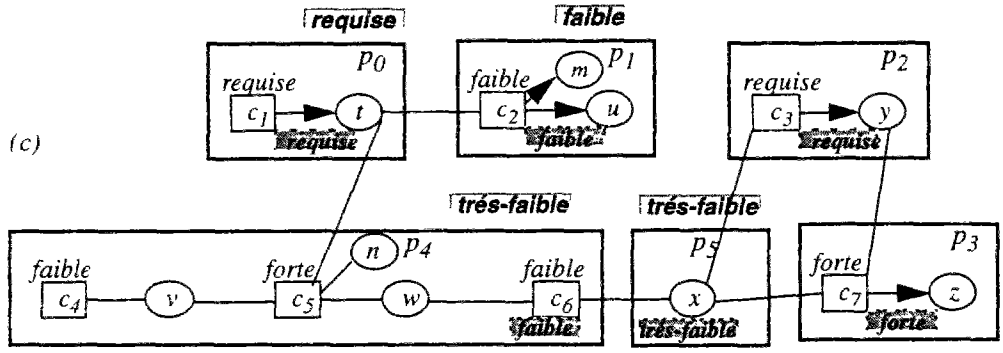
- Dans le but d'obtenir un graphe-solution, la cellule p_{23} est décomposée en trois cellules: p_2, p_3 et p_5. les *étiquettes-voyageuses* des cellules sont recalculées (figure 73.c).

- La cellule sur-contrainte p_4 est fusionnée à la cellule p_5 (puisqu'elles possèdent des étiquettes-voyageuses équivalentes) (figure 73.d). Le graphe obtenu satisfait les conditions d'un graphe solution.

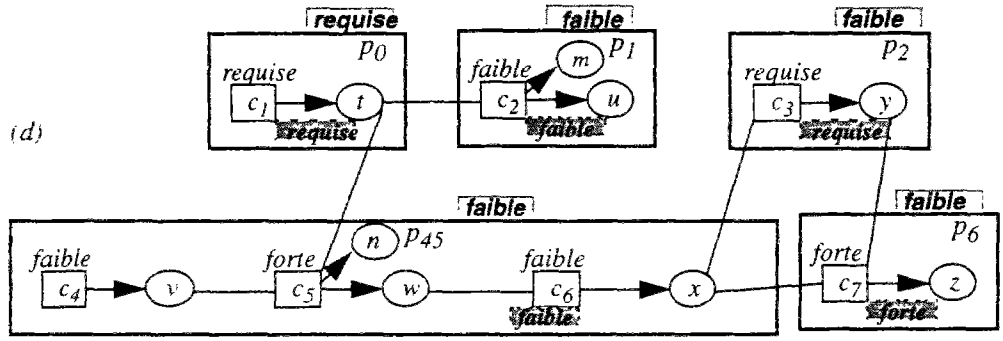
FIGURE 73 : Retrait d'une contrainte d'un graphe solution



(c)


requise
faible
requise
faible
requise
p_0
p_1
p_2
c_1
t
c_2
m
u
c_3
y
très-faible
très-faible
faible
forte
faible
p_4
p_5
p_3
c_4
v
c_5
n
w
c_6
x
c_7
z

(d)


requise
faible
faible
requise
faible
p_0
p_1
p_2
c_1
t
c_2
m
u
c_3
y
faible
faible
p_{45}
p_6
faible
forte
faible
c_4
v
c_5
n
w
c_6
x
c_7
z

Synthèse du chapitre

Comme dans les chapitres précédents dans ce mémoire, l'objet de ce chapitre a été de faire une analyse plus poussée de la théorie des hiérarchies de contraintes. Ici on considère des hiérarchies où les modes de combinaison peuvent varier selon les niveaux. Ce chapitre présente également une autre vue sur les graphes solutions à une hiérarchie de contraintes. La résolution de ces graphes solutions permet l'obtention des valuations de l'ensemble S qui sont des solutions à la hiérarchie de contraintes considérée.

Certains sous-ensembles de contraintes ne peuvent pas être résolus par une simple utilisation de l'algorithme de la propagation locale. Ces sous-ensembles de contraintes doivent être résolus ensemble. Par exemple, un sous-ensemble de contraintes dont les méthodes forment un circuit ou encore, un sous-ensemble de contraintes associées à un des deux critères globaux de comparaison (*MCM* ou *PCM*) ayant leurs méthodes en conflit. Le fait de considérer des cellules de contraintes permet de localiser ces sous-ensembles et donc de pouvoir les résoudre localement en utilisant des résolveurs spécifiques.

Un autre intérêt de la décomposition du graphe en cellules de contraintes est de ne pas être obligé de résoudre toutes les contraintes du système lors d'une perturbation. En moyenne, avec la décomposition, seul un sous-ensemble de cellules du graphe est réexaminé. Cependant, il n'est pas très intéressant d'utiliser cette modélisation pour des applications non incrémentales où l'utilisateur a besoin d'avoir une solution à son système de contraintes et non une solution à l'ajout de chaque contrainte. Puisque ici, pour chaque contrainte introduite, il y a une réorganisation d'un ensemble de cellules afin d'aboutir à un graphe-solution.

Ce chapitre présente une nouvelle modélisation des cellules de contraintes. Cette modélisation a été conçue dans le but de pouvoir manipuler des contraintes multi-sorties. Outre leur intérêt dans certains applications réelles [Ros94, San94, Hil93], elles permettent de donner un aspect optimal et élégant pour l'expressivité des contraintes du problème traité.

L'obtention d'un graphe solution après ajout ou retrait d'une contrainte est réalisée en considérant un graphe admissible de cellules de contraintes ainsi que des techniques d'optimisation qui sont basées sur la propagation locale (c.à.d *étiquette-voyageuse* et *étiquette-interne* d'une cellule). Ces dernières permettent de localiser les cellules à modifier en répercutant les effets de l'ajout ou du retrait d'une contrainte.

La résolution globale de la hiérarchie consiste en la résolution locale de chacune des cellules du graphe solution de cette hiérarchie et en la propagation des valeurs obtenues de cellule en cellule dans le graphe.

On a prouvé la correction et la terminaison de l'algorithme présenté dans ce chapitre. Cet algorithme constitue une simple modélisation afin de faire coopérer des résolveurs spécifiques. Ces résolveurs spécifiques doivent être conçus selon les instances de comparateurs (les critères) utilisés dans l'application réelle traitée.

L'utilisation de ce résolveur sur un graphe contenant des gros cycles n'est pas non plus très souhaitable puisque le graphe solution est réduit à un petit nombre de cellules contenant beaucoup de contraintes (du fait qu'on ne se permet pas d'avoir des dépendances cycliques entres les cellules) et donc il y a une perte d'efficacité au niveau de la résolution globale du graphe.

Le tableau de la figure 74 situe cet algorithme par rapport aux autres algorithmes vus dans les chapitres précédents:

FIGURE 74 : Comparaison entre résolveurs basés sur la propagation locale.

utilise / résolveurs	contraintes ont plusieurs variables en sortie	contraintes sont multi-directionnelles	contraintes sont dans une hiérarchie	compatible avec résolveur de cycle	utilise un critère global	unifie plusieurs types de comparateurs
Blue	–	+	–	–	–	–
DeltaBlue	+ –	– +	+	–	–	–
SkyBlue	+	+	+	+	–	–
QuickPlan	+	+	+	+	–	–
Houria	+	+	+	– +	+	–
LSCS	–	+	+	+	+	+
Latif	+	+	+	+	+	+

Conclusion

L'objectif de ce mémoire a été de :

1. *Définir le rôle d'une hiérarchie de contraintes.*

   Une hiérarchie de contraintes est une solution au problème de spécification déclarative de ce qu'il faut changer à la suite d'une perturbation introduite sur une contrainte du système.

   Le domaine d'application des hiérarchies de contraintes est tout genre d'application où l'on veut prendre en compte des contraintes de préférence comme par exemple dans la planification, l'ordonnancement ou encore la manipulation de figures géométriques.

   Une hiérarchie de contraintes permet de prendre en compte différents modes de combinaison des erreurs par niveau et utilise une agrégation globale de type lexicographique sur les valeurs de ces combinaisons.

2. *Concevoir un nouveau résolveur de maintien de solutions pour les hiérarchies de contraintes fonctionnelles.*

   Dans le formalisme des hiérarchies, les contraintes sont réparties par niveaux d'importance. Les valuations de l'ensemble des variables de ces contraintes sont triées par un ordre partiel défini à partir des erreurs (binaires ou numériques) et des mécanismes d'agrégation de ces erreurs (global, local, régional).

   Une solution à une hiérarchie de contraintes fonctionnelles qui utilise des erreurs binaires et un critère de comparaison est obtenue *via* la planification d'un graphe de méthodes de ces contraintes fonctionnelles et ensuite l'exécution de ces méthodes selon l'ordre topologique dans le graphe (d'où propagation locale). Le graphe de méthodes doit respecter le critère utilisé.

   Dans ce cadre, la plupart des travaux concernant la résolution des hiérarchies de contraintes fonctionnelles utilisent des erreurs binaires et un mode de comparaison local assez simple entre les différentes configurations.

   Nos travaux ont consisté en la conception d'un nouveau résolveur de propagation locale basé sur l'utilisation d'un algorithme du type "*meilleur d'abord*". Ce résolveur prend en compte différents modes de combinaison des erreurs par niveau et utilise une agrégation globale de type lexicographique sur les valeurs de ces combinaisons.

Le premier mode de combinaison global (le nombre de contraintes non satisfaites) que nous avons utilisé permet de trouver des valuations de meilleure qualité que celles obtenues par un mode de comparaison local.

Les deuxième et troisième modes de combinaison globaux (respectivement une combinaison où les poids représentent des prioritées, la somme des poids des contraintes non satisfaites) que nous avons intégrés consistent :

d'une part à surmonter la restriction imposée par les résolveurs existants et donc de prendre en compte des hiérarchies contenant des contraintes étiquetées et pondérées,

et d'autre part à permettre l'expressivité des contraintes de remplacement par niveau de la hiérarchie ainsi que des contraintes où le poids d'une contrainte traduit en quelque sorte son degré d'importance dans le niveau de la hiérarchie où elle se trouve.

L'intégration d'un autre mode de combinaison global défini sur les erreurs binaires ne doit pas poser de problèmes majeurs, il suffit de modifier la fonction d'évaluation utilisé.

3. *Proposer une procédure utilisant le résolveur précédent pour réaliser la comparaison inter-hiérarchies dans les Langages de Programmation Logique par Hiérarchie de Contraintes* .

La *PLHC* étend la *PLC* en incluant les hiérarchies de contraintes. La *PLHC* est paramétrée par le mode de combinaison *C* utilisé. L'expérience d'écrire des programmes en *PLHC* où *C* est associé à un mode de combinaison local montre que ce mode local produit des solutions non intuitives puisqu'un mode local ne peut comparer que des solutions résultant d'une seule hiérarchie (comparaison intra-hiérarchie).

Plusieurs applications pratiques en *PLHC* suggèrent le besoin d'une comparaison entre les solutions résultant des différents choix de règles du programme (comparaison inter-hiérarchies) afin d'éliminer celles qui sont non intuitives (c.à.d. jugées moins bonnes).

Pour réaliser une comparaison inter-hiérarchies, seuls les comparateurs globaux peuvent être utilisés. Le résolveur décrit précédemment implémente des comparateurs globaux et il peut donc être utilisé à cet effet.

On propose dans ce mémoire une procédure qui peut être incorporée dans les langages de *PLHC* afin de réaliser la comparaison inter-hiérarchies et donc de pouvoir éliminer les hiérarchies telles que leurs résolutions produiraient des solutions non désirables.

Une telle procédure est plus prometteuse qu'un algorithme du type séparation et évaluation en particulier lorsque le nombre de choix de règles alternatives d'un prédicat du programme est petit. Une évaluation expérimentale serait la bienvenue pour pouvoir déterminer jusqu'à quelle limite (c.à.d déterminer en moyenne une borne pour le nombre de choix de règles alternatives d'un prédicat du programme) cette procédure est intéressante à utiliser.

4. *Modéliser un résolveur pour la résolution de hiérarchies de contraintes où les modes de combinaison peuvent varier selon les niveaux. Ce dernier prend en compte des contraintes possédant des méthodes recalculant plusieurs variables à la fois.*

   Ici, on considère des hiérarchies où les modes de combinaison peuvent varier selon les niveaux. Un mode de combinaison global sur les erreurs du dernier niveau de la hiérarchie (ce mode peut être la somme des carrés des erreurs métriques ou le pire cas des erreurs métriques) et pour les autres niveaux, un mode de combinaison local sur leurs erreurs respectives.

   Certains sous-ensembles de contraintes ne peuvent pas être résolus par une simple utilisation de l'algorithme de la propagation locale et doivent être résolus ensemble. Le fait de considérer des cellules de contraintes permet de :

   localiser ces sous-ensembles et donc de pouvoir les résoudre localement en utilisant des résolveurs spécifiques.

   lors d'une perturbation, réexaminer seulement un sous-ensemble de cellules du graphe pour rétablir la consistance.

   On présente une nouvelle modélisation des cellules de contraintes afin de pouvoir utiliser des contraintes multi-sorties.

   L'algorithme proposé constitue une simple modélisation afin de faire coopérer des résolveurs spécifiques. Ces résolveurs spécifiques doivent être conçus selon les modes de combinaison utilisés.

   Des travaux futurs dans cette voie doivent définir la classe des modes de combinaison globaux d'erreurs qui se comportent de la même façon que les modes de combinaison considérés par ce résolveur, afin de pour pouvoir résoudre avec ce résolveur des hiérarchies ayant à leur dernier niveau ces modes de combinaisons.

Références

[BDF+87]Alan Borning, Robert Duiberg, Bjorn Freeman-Benson, Axel Kramer, and Michael Woolf. Constraint hierarchies. In ACM, editor, *proc. of the 1987 ACM Conference on Object-Oriented Programming Systems Languages, and Applications*, pages 48–60, October 1987.

[BFW92]Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, September 1992.

[BFW94]Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. *Constraint Programming*, computer and systems Constraint Hierarchies, pages 75–115. Number 131 in F. Springer-Verlag, 1994.

[BLB92]J. Bowen, R. Lai, and D Bahler. Fuzzy semantics and fuzzy constraint networks. In *proc. of the 1st IEEE Conference on Fuzzy Systems*, pages 1009–1016, San Fransisco, 1992.

[BMM+89]Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint hierarchies and logic programming. In *proc. of the Sixth International Conference on Logic Programming*, pages 149–164, Lisbon, June 1989.

[BN96] Mouhssine Bouzoubaa et Bertrand Neveu. Résolveurs de hiérarchies de contraintes. Dans les actes de la *Deuxième Conférence Internationale en Recherche Opérationnelle : Théorie et Applications (CIRO'96)*, Marrakech, Juin 1996.

[BNH95b]Mouhssine Bouzoubaa, Bertrand Neveu, and Geir Hasle. Houria II : A solver for hierarchical systems, planning of lexicographic satisfied count best case better graph for equational constraints. In *Constraint for Graphics and Visualization*, Cassis, France, September 1995. International Workshop in conjunction with CP-95.

[BNH95a]Mouhssine Bouzoubaa, Bertrand Neveu, and Geir Hasle. A solver for equational constraints in a hierarchical system. In *proc . of the Over Constrained Systems (OCS)*, Cassis, France, September 1995. International Workshop in conjunction with CP-95.

[BNH96]Mouhssine Bouzoubaa, Bertrand Neveu, and Geir Hasle. *Computer Science and Operations Research : Recent Advences in The Interface*, chapter Houria III : Planning of Lexicographic Weight Sum Better Graph for Equational Constraints. INFORMS, CSTS, Dallas, Texas, January 1996.

[Boh90]Karl-Friedrich Bohringer. Using constraints to achieve stability in automatic graph layout algorithms. In ACM SIGCHI, editor, *Proc. of CHI'90*, pages 43–52, Seattle, Washington, April 1990.

[Bor81] Alan H. Borning. The programming language aspects of Thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[Bou94]Mouhssine Bouzoubaa. Un résolveur de système de contraintes fonctionnelles hiérarchique. Rap. de Rech. 94-32, INRIA-CERMICS, Sophia-Antipolis, Novembre 1994.

[Bou95a]Mouhssine Bouzoubaa. The Houria constraint solver. In IOS press, editor, *Proc. of Fifth Scandinavian Conference On Artificial Intelligence (SCAI'95)*, Trondheim, Norway, May 1995.

[Bou95b]Mouhssine Bouzoubaa. The Houria constraint solver. In *proc. of the International Conference on Applications of Artificial Intelligence in Engineering X (AIEng'95)*, pages 219–226, Udine, Italie, July 1995.

[Bou95c]Mouhssine Bouzoubaa. An incremental constraint solver. In *proc. of Constraint'95*, pages 104–111, Florida, USA, April 1995. FLAIRS-95. International Workshop on Constraint-Based Reasoning.

[Bou95d]Mouhssine Bouzoubaa. Latif : Solver for multi-output constraints in hierarchical system. Draft paper, presented on the SINTEF International Symposium on Constraint Reasoning and Related Topics, June 1995.

[Bou96]Mouhssine Bouzoubaa. Functional constraint hierarchies in constraint logic programming. In Springer, editor, *proc. of the International Conference on the Principles and Practice of Constraint Programing (CP-96)*, Cambridge, Massachusetts, USA, August 1996.

[Bre89] J. Brewka. Preferred subtheories: an extended logical framework for default reasoning. In *proc. of the 11th. Joint Conference on Artificial Intelligence (IJCAI 89)*, pages 1043–1048, Detroit, MI, 1989.

[Coh90]Jacques Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68. July 1990.

[Col87] Alain Colmerauer. *An Introduction to PrologIII*. Groupe Intelligence Artificielle, Université Aix-Marseille II, Novembre 1987.

[DL85] R. Descottes and J. C. Latombe. Making compromises among antagonist constraints. *Artificial Intelligence*, 27:149–164, 1985.

[DVS+88]M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoum, T. Graf, and F. Bertheir. The constraint logic programming language Chip. In *proc. of FGCS-88*, 1988.

[Far94] Hélène Fargier. *Problème de satisfaction de contraintes flexibles application à l'ordonnancement de production*. thèse de doctorat, Institut de Recherche en Informatique de Toulouse, Juin 1994.

[Fre88] B. N. Freeman-Benson. Multiple solutions from constraint hierarchies. Technical Report 88-04-02, Department of Computer Science and Engineering, University of Washington, April 1988.

[Fre91] Bjorn N. Freeman-Benson. *Constraint Imperative Programming*. PhD thesis, Department of Computer Science and Engineering, University of Washington, July 1991. Published as Technical Report 91-07-02.

[FB92] Bjorn Freeman-Benson and Alan Borning. The design and implementation of Kaleidoscope'90, a constraint imperative programming language. In *proc. of the IEEE Computer Society International Conference on Computer Languages*, pages 174–180, April 1992.

[FM89] Bjorn Freeman-Benson and John Maloney. The Deltablue algorithm : An incremental constraint hierarchy solver. In IEEE, editor, *proc. of the Eighth Annual IEEE Phoenix Conference on Computer and Communications*, Scottsdale, Arizona, March 1989.

[FMB89]Bjorn Freeman-Benson, John Maloney, and Alan Borning. The Deltablue algorithm: An incremental constraint hierarchy solver. Technical Report 89-08-06, Department of Computer Science and Engineering,, University of Washington, August 1989.

[FMB90]Bjorn Freeman-Benson, John Maloney, and Alan. Borning. An incremental constraint solver. *Communication of the ACM*, 33(1):54–63, January 1990.

[FW90] Bjorn Freeman-Benson and Molly. Wilson. Deltastar, how i wonder what you are : A general algorithm for incremental satisfaction of constraint hierarchies. Technical Report 90-05-02, Department of Computer Science and Engineering, University of Washington, May 1990.

[FWB92]Bjorn Freeman-Benson, Molly Wilson, and Alan Borning. Deltastar: A general algorithm for incremental satisfaction of constraint hierarchies. In IEEE, editor, *the Eleventh Annual IEEE Phoenix Conference on Computers and Communications*, Scottsdale, Arizona, March 1992.

[FHS92]B. Faltings, D. Haroud, and I. Smith. Dynamic constraint propagation with continuous variables. In *proc. of the ECAI'92*, pages 754–758, 1992.

[FS89] E. C. Freuder and Snow. Improved relaxation and search methods for approximate constraint satisfaction with a maximum criterion. In *proc. of the 8.th Conference of the Canadian Society for Computational Studies of Intelligence*, pages 227–230, 1989.

[GF92] Q. Guan and G. Friedrich. Extending constraint satisfaction problem solving in structural design. In *proc. of the 5th International Conference IEA/AIE*, pages 341–350, Paderborn, Germany, June 1992.

[Gos83]James A. Gosling. *Algebraic Constraints*. PhD thesis, Carnegie-Mellon University, May 1983. Published as CMU Computer Science Departement tech report CMU-CS-83-132.

[GR92] Michel Gangnet and Burton Rosenberg. Constraint programming and graph algorithms. In *proc. of the Second International Symposium on Artificial Intelligence and Mathematics*, January 1992.

[HBR+94]Ralph D. Hill, Tom Brinck, Steven L. Rohall, John F. Patterson, and Wayne Wilner. The Rendezvous architecture and language for constructing multiuser applications. *ACM Transactions on Computer Human Interaction*, 1:81–125, June 1994.

[Hil93] Ralph D. Hill. The Rendezvous constraint maintenance system. In ACM SIGGRAPH SYMPOSIUM on User Interface Software and Technology, editors, *proc. of the UIST'93*, Atlanta, GA, November 1993.

[HJM+87]N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The clp(r) programmer's manual. Technical report, Computer Science Departement, Monash University, 1987.

[HKS+94]Hosobe Hiroshi, Miyashita Ken, Takahashi Shin, Matsuoka Satoshi, and Akinori Yonezawa. Locally simultaneous constraint satisfaction. In *proc. of the Principles and Practice of Constraint Programming*, LNCS 874, November 1994.

[HZ83] R. Hummel and S. Zucker. On the foundations of relaxation labeling processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3):267–287, 1983.

[Ign83] James P. Ignizio. Generalised goal programming. *Computers and Operations Research*, 10(4):277–290, 1983.

[Ign85] James P. Ignizio. Introduction to linear goal programming. Sage Publications, Beverly Hills, 1985. Sage University Paper Series on Qualitative Application in theorical Sciences, 07-560 Soc.

[Ilg87] ILOG, 2 av Galliéni, F-94250 Gentilly. *Le-Lisp*, v-15.21, Décembre 1987.

[JB96] Narendra Jussien est Partice Boizumault. CSP dynamiques et relaxation de contraintes. Dans les actes *de la deuxième conférence nationale en Résolution Pratique de Problèmes NP-Complets*, pages 135–149, Dijon, Mars 1996.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *proc. of of the Fourteenth ACM Principles of Programming Languages Conference*, January 1987.

[JM87] J. Jaffar and S. Michaylov. Methodology and implementation of a CLP system. In *proc. of the Fourth International Conference on Logic Programming*, pages 196–218, Melbourne, May 1987.

[KJ84] M. Konopasek and S. Jayaraman. *The TK!Solver Book.* Osborne/McGraw-Hill, Berkeley, CA, 1984.

[KK91] Tomihisa Kamada and Satoru Kawai. A general framework for visualizing abstract objects and relations. *ACM Transactions on Graphics*, 10(1):1–39, January 1991.

[Lel86] W. Leler. *Specification and Generation of Constraint Satisfaction Systems Using Augmented Term Rewriting.* PhD thesis, University of North Carolina at Chapel Hill, 1986.

[Lel87] William Leler. *Constraint Programming Languages.* Addison-Wesley, 1987.

[Mac77]Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1987.

[Mah87]Michael J. Maher. Logic semantics for a class of committed-choice programs. In *proc. of the Fourth Internationational Conference on Logic Programming*, pages 858–876, Melbourne, May 1987.

[Mal91]John. Maloney. *Using Constraints for User Interface Construction.* PhD thesis, Department of Computer Science and Engineering, University of Washington, August 1991. Published as Technical Report 91-08-12.

[Clo92] R. Martin-Clouaire. Dealing with soft constraints in a constraint satisfaction problem. In *proc. of the International Conference on Information Procesing of Uncertaintly in Knowledge-based Systems IPMU'92*, pages 37–40, Palma de Majorque, 1992.

[MGD+90a]Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Plippe Mar, and Ed. Pervin. The Garnet toolkit reference manuals: Support for highly-interactive graphical user interfaces in lisp. Technical Report CMU-CS-90-117, Computer Science Dept, Carnegie Mellon University, March 1990.

[MGD+90b]Brad A. Myers, Dario Guise, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Plippe Mar, and Ed. Pervin. Comprehensive support for graphical, highly-interactive user interfaces : The garnet user interface development environment. *IEEE Computer*, 11(23):71–85, November 1991.

[MM88]R. Mohr and G. Masini. Good old discrete relaxation. In *proc. of the ECAI'88*, pages 651–656, Munchen FRG, 1988.

[MS91] Michael J. Maher and Peter J. Stuckey. Expanding query power in constraint logic programming. Technical Report 91-06-13, Department of Computer Science and Engineering, University of Washington, 1991.

[Mur83]Katta G. Murty. *Linear Programming.* Wiley, 1983.

[Nel85] Greg. Nelson. Juno : A constraint-based graphics system. In ACM, editor, *proc . of the SIGGRAPH*, pages 235–243, July 1985.

[Rob66]A. Robinson. *Non-Standard Analysis.* North-Holland Publishing Company, Amsterdam, 1966.

[San93] Michael Sannella. The Skyblue constraint solver. Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington, February 1993.

[San94] Michael Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces.* PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, 1994. Also available as Technical Report 94-09-10.

[Sar85] Vijay A. Saraswat. Problems with concurrent prolog. Technical Report CS-86-100, Carnegie-Mellon University, May 1985. Revised January 1986.

[Sar89] Vijay Anand Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Computer Science Departement, January 1989.

[Sch92] Thomas Schiex. Possibilistic constraint satisfaction problems or "how to handle soft constraints". In *proc. of the Eighth Conference on Uncertaint in Artificial Intelligence*, pages 268–275, Stanford, CA, July 1992.

[Sha86] Ehud. Shapiro. Concurrent prolog : A progress report. *IEEE Computer*, 19(8):44–58, August 1986.

[Sis90] Steven Sistare. *A Graphical Editor for Constraint-Based Geometric Modeling*. PhD thesis, Departement of Computer Science, Harvard, December 1990. Published as Technical Report TR-06-9.

[SJ92] Thomas Schiex et Philippe Janssen. Représentation et traitement pratique de la flexibilité dans les problèmes sous contraintes. Dans les actes des *Journées Nationales du PRC-IA*, pages 369–426, Marseille, Octobre 1992.

[SRP91] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *proc. of the Eighteenth Annual Principles of Programming Languages Symposim ACM*, 1991.

[SSA91] Takahashi Shim, Matsuoka Satoshi, and Yonezawa Akinori. A general framework for bi-directional translation between abstract and pictorial data. In ACM, editor, *proc. of the ACM SIGGRAPH Symposium on User Interface Software and Technyolog*, pages 165–174, Hilton Head, South Carolina, November 1991.

[Sut63b] Ivan Sutherland. *Sketchpad : A Man-Machine Graphical Communication System*. PhD thesis, Departement of Electrical Engineering, MIT, January 1963.

[Sut63a] Ivan Sutherland. Sketchpad: A man-machine graphical communication system. In *proc. of the Spring Joint Computer Conference, IFIPS*, 1963.

[SFV95] Thomas Schiex, Hélène Fargier and Gerard Verfaillie. Valued constraint satisfaction problems : hard and easy problems. In *proc. of the* IJCAI, pages 631-637, CA, 1995.

[SV94] Thomas Schiex and Gerard Verfaillie. Nogood recording fot static and dynamic constraint satisfaction problems. *Artificial Intelligence Tools*, 3(2):187–207, 1994.

[Tro95] Gilles Trombettoni. Formalizing local propagation in constraint maintenance systems. In Springer, editor, *proc. of EPIA '95*, number 990 in LNAI, Madere, Portugal, 1995.

[Van89] Pascal Van Hentenryck. Constraint satisfaction in logic programming. *MIT Press*, 1989. Cambridge, MA.

[VS95] Gerard Verfaillie et Thomas Schiex. Maintien de solution dans les problèmes dynamiques de satisfaction de contraintes : bilan de quelques approches. Revue d'*Intelligence Artificielle*, 9(3), 1995.

[Van88] Brad Vander Zanden. *Incremental Constraint Satisfaction and Its Application to Graphical Interfaces*. PhD thesis, Department of Computer Sciences, Cornell University, Ithaca, NY, 1988.

[Van92] Brad Vander Zanden. A domain-independent algorithm for incrementally satisfying multi-way constraints. Technical Report CS-92-160, Department of Computer Science, University of Tennessee, July 1992.

[Van95]Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way, dataflow constraints. Technical Report CS-95-282, Computer Science Department, University of Tennesse, Knoxville TN 37996, March 1995.

[VMG+94]Brad Vander Zanden, A. Brad Myers, Dario Giuse, and Pedro Szekely. Integrating pointer variables into one-way constraint models. *ACM Transactions on Computer Human Interaction*, 1:161–213, June 1994.

[Wal75]D. Waltz. *The psychology of computer vision*. Winston, Mac-Graw Hill, NY, 1975.

[WB89]Molly Wilson and Alan Borning. Extending hierarchical constraint logic programming: Nonmonotonicity and inter-hierarchy comparaison. In *proc of the North American Conference on Logic Programming*, Cleveland, October 1989.

[Wil92]Molly Wilson. *Hierarchical Constraint Logic Programming*. PhD thesis, Department of Computer Science and Engineering, University of Washington, 1992.

[Wil94]Rosener. J. William. *Integrating Multi-way and Structural Constraints into Spreadsheet Programming*. PhD thesis, Department of Computer Science. University of Tennessee, Knoxville, TN, 1994.

[Wyk80]Christopher J. van Wyk. *A Language for Typesetting Graphics*. PhD thesis, Department of Computer Science, Stanford, June 1980.

[Wyk82]Christopher J. van Wyk. A high-level language for specifying pictures. *ACM Transactions on Graphics*, 1(2), April 1982.

Résumé

L'objectif de ce travail est de proposer quelques approches pour la résolution de hiérarchies de contraintes fonctionnelles.

Dans un premier temps, le rôle et les qualités d'une hiérarchie de contraintes sont définis. Une hiérarchie de contraintes permet de résoudre des problèmes sur-contraints en répartissant les contraintes dans une hiérarchie (de niveaux) suivant leur importance.

Dans un deuxième temps, un nouveau résolveur de maintien de solutions pour les hiérarchies de contraintes fonctionnelles a été conçu afin d'obtenir des solutions de meilleure qualité. Ce résolveur est basé sur l'utilisation d'un algorithme du type "*meilleur d'abord*" et prend en compte différents modes de combinaison des erreurs par niveau et utilise une agrégation globale de type lexicographique sur les valeurs de ces combinaisons. Les modes de combinaison globale intégrés dans ce résolveur sont : le nombre de contraintes non satisfaites, une combinaison où les poids représentent des priorités pour considérer des contraintes de remplacement et enfin la somme des poids des contraintes non satisfaites.

Dans un troisième temps, nous proposons une procédure utilisant le résolveur précédent. Cette dernière est plus prometteuse qu'un algorithme du type séparation et évaluation. Elle peut être incorporée dans les langages de Programmation Logique par Hiérarchie de Contraintes afin de réaliser la comparaison inter-hiérarchies et donc de pouvoir éliminer les hiérarchies telles que leurs résolutions produiraient des solutions non désirables.

Enfin, nous avons modélisé un résolveur pour la résolution de hiérarchies de contraintes où les modes de combinaison peuvent varier selon les niveaux. Ce dernier prend en compte des contraintes possédant des méthodes recalculant plusieurs variables à la fois et établit un plan de coopération entre des résolveurs spécifiques. Ces résolveurs spécifiques doivent être conçus selon les modes de combinaison utilisés.

Mots-clés : contraintes, contraintes fonctionnelles, hiérarchie de contraintes, propagation locale, programmation logique par hiérarchie de contraintes, comparaison inter-hiérarchies.

Abstract

The goal of this work is to propose some approaches that solve functional constraint hierarchies.

First, we define the role and the interesting features of a constraint hierarchy. A constraint hierarchy allows to represent and solve over-constrained problems.

Second, we have conceived a new solver that solves and maintains solutions of a functional constraint hierarchy, in order to obtain solutions of a better quality than the existing solvers. The algorithmic approach used by this solver is the "*best first search*". This solver considers different error combination modes at each level of the hierarchy and uses a global lexicographic aggregation on the values of these combinations. The global combination modes integrated in this solver are : the number of unsatisfied constraints, a combination where the weights represent priorities between the constraints in order to handle replacement constraints and the sum of the weights of unsatisfied constraints.

Third, we propose an algorithm based on the previous solver and on the extended notion of comparators for comparisons between the hierarchies that arise from alternate rule choices in the program. This algorithm is more promising than a branch and bound algorithm and can be incorporated in hierarchical constraint logic programing languages. This algorithm allows to achieve the inter-hierarchies comparison and to eliminate the hierarchies such that these hierarchies may lead to undesirable solutions.

Finally, we propose a model of a second solver. It treats a single system where different types of constraints are unified. This solver supports constraints with multi-output variables. If constraints don't need to be solved simultaneously then this solver divides them as much as possible into subsets of constraints and solves each subset by invoking a specific subsolver.

Keywords: constraints, functional constraints, constraint hierarchy, local propagation, hierarchical constraint logic programing, inter-hierarchies comparison.